SynxDB Elastic Documentation

Release Preview

Synx Data Labs, Inc

Aug 21, 2025

Table of Contents

Disclaimer

Greenplum® is a registered trademark of Broadcom Inc. Synx Data Labs and SynxDB are not affiliated with, endorsed by, or sponsored by Broadcom Inc. Any references to Greenplum are for comparative, educational, and interoperability purposes only.

Chapter 1

Product Overview

1.1 Product Overview

SynxDB Elastic is a cloud-native, distributed analytical database that offers enterprises efficient, flexible, and scalable solutions for data management and analysis. Built on Apache Cloudberry and containerized, it separates storage, computing, and metadata to maximize the benefits of cloud-native infrastructure. This enables businesses to integrate their data resources, store and analyze massive datasets, and use data to drive business growth.

To have a quick try of SynxDB Elastic, see *Quick-Start Guide*.

1.2 Key Concepts

SynxDB Elastic employs state-of-the-art cloud-native architecture. The following sections outline the key concepts in SynxDB Elastic to help you understand its components and functionalities.

1.2.1 Organizations

An organization is the highest-level administrative unit in SynxDB Elastic, responsible for managing resources and billing. Organization users are administrators with permission to manage the organization's resources.

1.2.2 Accounts, User, and Warehouses

Organizations in SynxDB Elastic are divided into logical subdivisions called accounts. Each account serves as an isolated environment that contains its own data and compute resources, including users and warehouses.

Accounts

Each account is a distinct logical resource unit within an organization, managing its own database data and compute resources without access to those of other accounts. This structure ensures clear boundaries between accounts while enabling flexible resource management within each account. Organization users can create or delete accounts as needed. Each account is assigned a unique host address (domain name) for access, including connections via psql.

User

Users are defined at the account level, connecting to one or more databases and functioning similarly to PostgreSQL or Apache Cloudberry users. Each user represents an individual or system that interacts with databases within an account to perform operations such as querying, updating, or managing data. Users can be created, managed, and deleted via the console or the psql client, assigned roles and privileges to access data and warehouses in the account. Each user belongs exclusively to one account and is managed at the account level, ensuring clear access control and security.

Warehouses

Warehouses are compute engines within an account to process queries. Each warehouse is composed of multiple segments, which can be scaled horizontally as needed. All warehouses within a single account have access to a shared pool of data and perform computing tasks independently. Key features of warehouses include:

- Resource Management: Allows pausing, resuming, and resizing on demand.
- Scaling: Support changing segment counts for online horizontal scaling.

1.2.3 Object Storage

Object storage is a scalable and durable cloud storage solution. In SynxDB Elastic, it acts as the persistent storage layer for large datasets, enabling data loading, backups, and external table storage. Its flexible design ensures secure, cost-effective storage for both structured and semi-structured data.

1.3 Architecture and Services

SynxDB Elastic employs a cloud-native, storage-compute separated architecture that consists of four distinct layers: the metadata service layer, compute service layer, storage service layer, and data science and AI services layer. This design enables efficient resource utilization, horizontal scalability, and optimal performance for diverse analytical and machine learning workloads.

1.3.1 Architecture Overview

The four-layer architecture provides clear functional separation and enables independent scaling of each component:

- **Metadata Service Layer**: Manages metadata persistence and access, handles transaction management and state persistence
- **Compute Service Layer**: Processes query requests with coordinator nodes handling query optimization and segment nodes executing queries, supports multiple warehouses for independent compute clusters
- **Storage Service Layer**: Provides data persistence and access, supporting both row storage and column storage engines for different workload scenarios
- **Data Science and AI Services**: Provides comprehensive machine learning and AI capabilities through the SynxML Platform

1.3.2 Metadata Service Layer

SynxDB Elastic offers two metadata service options: FoundationDB (FDB)-based metadata service and UnionStore-based metadata service.

FoundationDB-based Metadata Service

The Metadata Service is an independent cluster built upon FoundationDB, specifically engineered to manage the diverse metadata required by a computing cluster. This service implements a compute-storage separation design principle, isolating metadata management operations. This architectural choice not only reduces the overall system coupling but also significantly enhances scalability. Its underlying storage leverages FoundationDB, an industry-recognized, highly available, distributed key-value (KV) database renowned for its exceptional scalability and reliability. FoundationDB supports both independent and multi-tenant deployment models, thereby balancing security requirements with cost-effectiveness.

The overall Metadata Service architecture comprises three core components, which collaborate to deliver its functionality:

- **Coordination and Session Management Service**: A session management cluster responsible for service discovery and the allocation of metadata service nodes.
- **Metadata Service**: A stateless service cluster providing critical metadata access, access control, lock management, and distributed transaction capabilities. Internally, this service is further modularized into specific functional components:
    - Object Definition Manager: Manages queries, modifications, and deletions of database object definitions (e.g., tables, views) and handles their associated access permissions.
    - Manifest Manager: Provides multi-version management for data manifest files.
    - Sequence Server: Offers sequence value allocation services.
    - WAL Manager: Handles the storage and application of metadata Write-Ahead Logs (WAL).
    - Concurrent Controller: Provides lock-based concurrency control to ensure data consistency during concurrent access.
- **Metadata Storage Service**: A distributed key-value storage engine implemented using FoundationDB, responsible for the actual storage of metadata. This includes, for example, mappings from tables to storage objects, data dictionaries, WAL logs, and index information.

Operational Workflow

Upon receipt of a metadata request, the Coordination and Session Management Service allocates an appropriate Metadata Service node to process the request. The designated node then retrieves the necessary metadata from the Metadata Storage Service and returns the results to the requester. Furthermore, the Coordination and Session Management Service functions as the connection entry point for Database Proxy nodes, managing communication with other nodes within the system.

UnionStore-based Metadata Service

UnionStore serves as an alternative metadata service, providing data persistence and access capabilities. The core concept is “log is database”. It persists redo logs and provides external access through log replay. UnionStore consists of two main components:

WAL Service

- Receives data logs from the compute layer and persists them with 3-replica reliability
- Synchronizes persisted logs to the Page Service

Page Service

- Retrieves and parses persisted logs from WAL Service
- Applies log modifications to corresponding pages to obtain the latest page versions
- Provides page read requests, returning the appropriate version of pages to clients

UnionStore Features

- **Multi-version Data**: Supports accessing any version of retained data (Time Travel capability)
- **Multi-tenancy**: A single metadata service cluster can serve multiple users, improving resource utilization and cost-effectiveness
- **Backup and Recovery**:
  - WAL Service backs up persisted logs to object storage for enhanced reliability and cost reduction
  - Page Service backs up current data snapshots to object storage as checkpoints
  - Recovery involves retrieving the latest snapshot from object storage and synchronizing data from the snapshot's log position

Using UnionStore as Metadata Service

- All metadata modifications and transaction information are persisted to UnionStore through WAL
- Each cluster uses a single tenant, with all cluster nodes accessing the tenant's metadata (including system table data and transaction data)
- Cluster nodes maintain metadata caches, accessing the metadata service only on cache misses

1.3.3 Compute Service Layer

The compute layer consists of two primary components: Coordinator and Segment nodes, where segments are the minimum units of a compute cluster (Warehouse).

Coordinator (Proxy)

Coordinator nodes primarily handle client request reception, query optimization, and query plan distribution to segments for execution.

Key Features

- **Metadata Distribution**:
  - Coordinator nodes distribute all relevant metadata along with query plans, avoiding segment access to metadata services
  - Significantly reduces metadata service pressure
  - Eliminates network interaction delays with metadata services
- **Lightweight and Stateless**:
  - Primarily handles query optimization and forwarding, with minimal computation involvement
  - Metadata access primarily uses cache
  - Metadata and data are stored separately, enabling fast startup and reconstruction

Warehouse

A Warehouse is a compute cluster composed of one or more segments, responsible for receiving and executing query plans from the coordinator and returning results.

Key Features

- **Multiple Warehouses**:
  - Supports launching multiple independent warehouses that share the same data while performing concurrent read/write operations
  - Different warehouses can be configured with varying specifications and scales for different business workloads on the same data

- **Stateless and Rapid Scaling**:
  - Warehouses can quickly scale segments without business impact to improve query performance
  - Warehouses can be created, started, and stopped rapidly (within seconds) based on business requirements, improving compute resource utilization

1.3.4 Storage Service Layer

The storage service layer supports both row storage and column storage engines to accommodate different workload scenarios. Data is persisted to:

1. **PAX Column Storage (Object Storage)**: Provides unlimited expansion capabilities
2. **UnionStore**: Supports fast row storage read/write operations

All cluster nodes maintain data caches for efficient data access.

UnionStore Row Storage

UnionStore serves as a row storage engine that can provide storage and access for user row tables in addition to metadata services. Users can create and use row tables through specific syntax.

Data Organization

- Data is organized at the tuple level and sequentially written to different pages (8K/32K)
- Page headers contain pointers for quick tuple location

Key Benefits

- Fast reading and writing of complete records
- More efficient DML operations
- Supports OLTP (Online Transaction Processing) workloads with frequent DML operations and high latency requirements
- Handles continuous, batch data writing scenarios
- Compatible with PostgreSQL ecosystem, supporting special requirements such as PostGIS

PAX Column Storage

PAX is a new column storage engine that organizes data by columns in continuous storage, grouping multiple column data into row groups within files stored in object storage.

Key Features

- **Statistical Information**:
  - Collects file-level and row group-level statistics during data writing to accelerate queries
  - Uses statistics as query results to avoid data scanning and computation (e.g., for aggregates)
  - Employs min-max filtering for file-level and row group-level data filtering during scanning
- **Vectorized Engine**:
  - Integrates with vectorized engines, generating internal column storage formats for enhanced data processing and query capabilities
- **Data Writing and Updates**:
  - Uses append-only writing for data insertion
  - Employs visibility map (visimap) for delete marking
  - Data reading determines record visibility based on visimap
  - Visimap is stored with data rather than as metadata, avoiding excessive metadata growth and frequent metadata access during data reads
  - Provides Copy on Write writing strategy for delete operations, rewriting related data files to avoid visimap generation, suitable for read-heavy, write-light scenarios
- **Data Clustering**:
  - Groups "adjacent data" together to enable more effective data filtering through min-max filters
- **File Management**:
  - Merges multiple data writes into single files to avoid small file generation

    - Periodically deletes unnecessary files and merges small files to improve data scanning efficiency

Gopher Cache

Gopher is a distributed cache system that supports high-concurrency, low-latency access to massive small and large files. It is deployed to each node in the data warehouse compute layer to accelerate computing I/O access.

- Adopts a decentralized design where each cache instance is a self-contained client/server service
- Acts as a local file system with two-level caches for data warehouse computing services
- Avoids potential intra-cluster network communication constraints observed in centralized master-slave distributed cache systems

Core Components

1. **Metadata Service**: The core of the cache system, managing metadata such as task status, file names, and block IDs, interfacing with other modules to facilitate client data access
2. **User File System**: Stores destination storage information including target file systems, ports, and buckets
3. **Block Manager and File Manager**: Provide data access for blocks and files respectively, utilizing stream reads and writes for large file processing in small memory buffers
4. **Cache Manager**: Implements a two-level cache (memory pool and SSD cache) to efficiently utilize limited cache space and improve data access speed
5. **Persistent Storage Module**: Incorporates network transmission libraries for S3 object storage, HDFS, and local storage connections
6. **Session Manager**: Utilizes epoll multiplexing to handle multiple client connections simultaneously

Key Features

- File prefetching, batch reading, shared memory, asynchronous write optimization
- Concurrent multiple reads and small file memory merging
- High-performance and highly available caching system supporting efficient concurrent processing and rapid scaling

1.3.5 Data Science and AI Services

SynxDB Elastic includes the SynxML Platform, a high-performance distributed computing platform designed for enterprise-grade AI applications. The platform integrates the strengths of open-source technologies and enhances them with enterprise-grade features, offering a complete end-to-end solution covering data preparation, feature engineering, model training, and production deployment.

SynxML Platform Core Features

Visual Modeling

- Offers an intuitive graphical interface that lets users train models by selecting algorithms, datasets, and setting hyperparameters
- After training, models can be deployed as services with a single click
- Supports A/B testing and staged rollouts

Notebook-based Development

- Comes with a built-in professional-grade Jupyter environment
- Developers can write and debug Python code flexibly, enabling customization and experimentation with complex algorithms

Algorithm and Model Library

- Includes a rich collection of built-in algorithms such as Logistic Regression and XGBoost
- Deep learning models like MLP, ResNet, and Bert
- Fine-tuning algorithms for large language models (supporting QLoRA and Unsloth)

- Users can also extend the library with custom algorithms

SynxML Platform Use Cases

Private Model Training for Enterprises

- Enables organizations to train machine learning models on sensitive internal data without leaving the enterprise network
- Ensures secure end-to-end management and high-performance training
- Helps businesses build accurate, compliant, and domain-specific models

Fine-tuning Large Language Models

- Integrates efficient fine-tuning methods such as QLoRA and Unsloth to support popular LLMs
- Enterprises can quickly adapt LLMs to specialized domains using few-shot learning
- Reduces training costs while improving task accuracy and semantic understanding

Model Deployment

- Allows trained models to be deployed as RESTful API services in one click
- Seamlessly integrates with business systems
- Supports A/B testing, phased rollouts, and auto-scaling to maintain stable performance under high concurrency

1.4 Architectural Principles

A key architectural distinction of SynxDB Elastic from MPP (Massively Parallel Processing) deployments is its mirror-less design. In traditional systems, mirroring is a primary mechanism for fault tolerance, where each primary segment has a corresponding mirror segment on a different host to ensure data redundancy.

SynxDB Elastic adopts a cloud-native architecture that leverages the inherent resilience and scalability of modern cloud infrastructure, making segment-level mirroring unnecessary. Here's why this design is advantageous:

1.4.1 Rationale for Not Using Mirrors

- **Cloud-native Resilience:** SynxDB Elastic is designed to run on resilient cloud infrastructure where data is typically stored in highly durable object storage services (like Amazon S3, Google Cloud Storage, or Azure Blob Storage). These services provide built-in data redundancy and fault tolerance across multiple availability zones, making segment-level mirroring redundant.
- **Simplified Architecture:** Eliminating mirrors simplifies the cluster architecture, reducing the number of segment instances to manage and lowering operational complexity. This leads to easier deployment, scaling, and maintenance.
- **Cost-Effectiveness:** A mirror-less design greatly reduces infrastructure costs. Because each primary segment does not require a dedicated mirror, the total number of virtual machines or physical hosts is halved. In addition, leveraging S3-compatible object storage is considerably less expensive than traditional cloud provider attached storage options (like AWS EBS or Azure Managed Disks). This strategic design choice leads to savings in compute and storage resources.

1.4.2 High Availability and Data Protection

In the absence of mirrors, SynxDB Elastic ensures high availability and data protection through a combination of mechanisms:

- **Durable Object Storage**: All data is stored in cloud object storage, which is inherently designed for high durability and availability. This means that even if a compute node fails, the data remains safe and accessible.
- **Stateless Compute Nodes**: The compute nodes (segments) in SynxDB Elastic are stateless.

If a node fails, the system can quickly provision a new one, which then attaches to the shared object storage and resumes processing. This results in faster recovery times compared to traditional mirror failover processes.

- **Coordinator High Availability**: The coordinator, which is the brain of the cluster, can be configured for high availability using a standby coordinator, ensuring that there is no single point of failure at the control plane level.

1.4.3 Performance and Cost Implications

The mirror-less design has several positive implications for performance and cost:

- **Improved Write Performance**: Without the need to synchronously write data to a mirror segment, the write throughput of the system is improved. This is particularly beneficial for write-intensive workloads.
- **Reduced Network Traffic**: The absence of mirroring reduces the network traffic between primary and mirror segments, freeing up bandwidth for query processing and other operations.
- **Lower Total Cost of Ownership (TCO)**: As mentioned earlier, the reduction in infrastructure requirements leads to a lower TCO. This allows organizations to allocate their budget to other critical areas.

1.5 Highlighted Features

SynxDB Elastic stands out in data management and analysis with its robust features.

1.5.1 Elastic architecture with separation of storage and computing

SynxDB Elastic's separation of storage and computing overcomes traditional MPP database limitations. Its stateless computing clusters can be created on demand, offering high elasticity and intelligent resource scheduling based on business load. Storage and computing can scale independently, eliminating data migration and enabling maintenance without downtime.

1.5.2 Comprehensive security measures

With a complete security mechanism, SynxDB Elastic ensures data safety through dynamic desensitization, unified authentication, on-demand authorization, and secure storage. It supports multiple encryption and desensitization algorithms to meet diverse security needs across development, testing, and sandbox environments.

1.5.3 Multi-source data and big data integration

SynxDB Elastic delivers robust data integration, empowering enterprises to build a hybrid data ecosystem. It leverages external tables and connectors to virtualize data, supports batch and real-time data collection from multiple sources, and deeply integrates with the big data ecosystem, including Kafka, HDFS and Hive.

In addition, SynxDB Elastic seamlessly integrates with a wide array of data sources, computing engines, data management tools, analysis tools, and programming languages, creating a comprehensive ecosystem. This seamless integration reduces costs and simplifies the adoption of SynxDB Elastic 4.0 into existing tech stacks, facilitating a lakehouse integrated platform for multi-source analysis without data migration.

1.5.4 Diverse analytical and computing capabilities

SynxDB Elastic offers rich analytical functions for descriptive, operational, and predictive analysis. It supports statistical analysis, machine learning, unstructured data analysis, HTAP, deep learning, vector search, spatiotemporal analysis, and UDF custom analysis to meet various business needs.

1.5.5 Containerized deployment

SynxDB Elastic supports containerized deployment with a cloud-native design. Divided into Control and Data Panels, it offers multi-tenant management, multi-dimensional scaling, and multi-platform adaptation. This enhances resource efficiency and deployment flexibility.

1.5.6 Product-level disaster recovery

SynxDB Elastic uses built-in CDC technology to synchronize metadata and business data remotely, ensuring high availability and disaster recovery. It replays synchronized transactions and Delta files in disaster recovery environments to maintain data consistency and integrity, minimizing data loss risks.

1.6 User Scenarios

SynxDB Elastic is a cloud-native, high-performance analytical database designed for large-scale data processing. It leverages a storage-compute separation architecture to provide elastic scalability, multi-modal analytics, and optimized performance for diverse workloads. Built on Apache Cloudberry, it extends the capabilities of traditional MPP databases with enhanced high availability, workload isolation, and automatic scaling.

This document describes the key use cases of SynxDB Elastic.

1.6.1 Data warehousing

A data warehouse is the crucial system for enterprise data analyses. SynxDB Elastic offers comprehensive enterprise-level data storage, management, and analysis capabilities. It supports PB-scale large datasets and complex SQL queries, aiding enterprises in data-driven decision support.

- Offline batch processing: Supports multiple methods to batch load source data into the data warehouse, creating operational data stores (ODS), data warehouse details (DWD), and data warehouse services (DWS). It builds source models and normalized models, including fact tables and dimension tables.
- Data marts: Processes different types of data to provide customized data sets for specific domains or departments.
- Business Intelligence (BI) reports and analyses: Handles complex data analyses and query needs, including data aggregation, multi-dimensional analyses, and associative queries. Supports business analyses, report generation, and decision support.

1.6.2 Big data/data lake integration and analyses

Big data platforms and data lakes are key infrastructures for enterprise data management. They help enterprises effectively integrate and utilize data resources, uncover data value, and support operational optimization and business expansion, making them essential systems for maintaining competitiveness.

SynxDB Elastic features a unified lakehouse architecture, serving as an integrated query engine on a big data platform for efficient exploration and analyses of structured, semi-structured, and unstructured data in data lakes.

- ETL batch processing: Integrates with various mainstream ETL tools for batch extracting, transforming, and loading external data sources.
- Lakehouse/multi-source joint analyses: Supports building a unified lakehouse, sharing metadata between data lakes and data warehouses, and enabling efficient data access. Facilitates joint analyses across different data sources for more comprehensive and in-depth insights.
- Interactive query analyses: Allows users to interact with data sets in real-time for exploration and analyses.
- GIS spatiotemporal data analyses: Analyzes geospatial and time-series data using Geographic Information System (GIS) formats to reveal spatial and temporal relationships.
- Log analyses: Stores, processes, and analyzes system log data to monitor and maintain system stability and security, and to optimize performance.

1.6.3 Real-time data analysis

In scenarios like mobile internet, IoT, and financial risk control, where quick responses are essential, data analysis systems must support low-latency decision-making. SynxDB Elastic integrates streaming data ingestion with hybrid transactional and analytical processing (HTAP), offering real-time data operations for insertions, deletions, and updates, as well as instant analysis of incremental data. This enables rapid data value extraction and real-time decision-making.

1.6.4 Generative AI data application development

Generative AI (GenAI) can create new content, such as text, charts, based on existing data or specific patterns, offering broad applications and potential value across multiple fields. The SynxDB Elastic + SynxML AI solution provides end-to-end capabilities for developing data intelligence applications based on GenAI large models, supporting the entire lifecycle from data storage to AI application deployment. Key capabilities include:

- Unstructured data management and analysis: Manages and processes unstructured data, including text, images, and audio, in a structured and unified manner.
- Vector knowledge base: Supports distributed storage and retrieval of high-dimensional data, building vector-based knowledge bases, and providing efficient retrieval-augmented generation (RAG) services.
- Model fine-tuning and post-pretraining: Supports full-parameter fine-tuning and LoRA

fine-tuning with multi-machine, multi-GPU setups, and mixed precision with parallel post-pretraining.

- Model inference and elastic deployment: Supports inference with multiple large models such as LLaMA, GLM, and DeepSpeed. Implements hybrid deployments of multi-node CPU + GPU servers with automatic scaling based on load.
- Large-model data intelligence applications: Supports data intelligence applications using GenAI large models, including natural language interaction analysis, document AI, and enterprise knowledge bases.

1.6.5 Data mining and machine learning

SynxDB Elastic, combined with SynxML, supports a wide range of data mining and machine learning algorithms. All algorithms run efficiently in a distributed manner within the database, eliminating the need for data movement or cross-platform management. The algorithm platform addresses data analysis needs for enterprise clients, including marketing, customer retention, personalized services, risk control, and supply chain management. Key capabilities include:

- Data mining: Supports common data mining algorithms such as prediction, clustering, association, text mining, sequence pattern analysis, anomaly detection, and network mining.
- Machine learning: Supports popular machine learning algorithms, including supervised learning, unsupervised learning, and reinforcement learning.
- Custom function extension: Allows users to write custom functions (UDFs) in languages such as R, Python, Perl, Java, and pgSQL to meet specific business analysis needs.

1.7 Comparison with Apache Cloudberry

Dimension	Apache Cloudberry	SynxDB Elastic
Architecture	MPP (Massively Parallel Processing), tightly coupled compute and storage	Separation of compute and storage, stateless compute layer with elastic scaling
Database kernel	Based on PostgreSQL 14, with enhancements	Uses Apache Cloudberry as the database kernel, modified for cloud-native architecture
Scalability	Fixed cluster, scaling requires data redistribution	Elastic scaling, compute and storage layers expand independently
Use cases	Suitable for on-premises big data analytics	Suitable for large-scale commercial customers, especially in cloud environments
Deployment model	Traditional on-premises deployment	Kubernetes deployment
Open-source / Commercial	Open-source, managed by Apache Foundation	Commercial, closed-source with additional enterprise features

1.8 Comparison with Greenplum

SynxDB Elastic is highly compatible with Greenplum, and provides all the Greenplum features you need.

In addition, SynxDB Elastic possesses some features that Greenplum currently lacks or does not support. More details are listed below.

1.8.1 General features

Note

- In the following tables, ✓ means support, and ✗ means no support.
- The feature comparison in the following tables is based on Greenplum 7 Beta.3.

Feature names	SynxDB Elastic	Greenplum
`EXPLAIN` (WAL) support	✓	✗
Multiranges	✓	✗
The `range_agg` range type aggregation function	✓	✗
`CREATE ACCESS METHOD`	✓	✓ (Upcoming)
LZ4 compression for `TOAST` tables	✓	✗
JSONB subscripting	✓	✗
Configure the maximum WAL retention for replication slots	✓	✗
Verify backup integrity (`pg_verifybackup`)	✓	✗
Clients can require `SCRAM` channel binding	✓	✗
Vacuum “emergency mode”	✓	✗
Certificate authentication with `postgres_fdw`	✓	✗
`UPSERT`	✓	✓ (Upcoming)
`COPY FROM WHERE`	✓	✗
`VACUUM` / `ANALYZE` Skip Lock Table	✓	✗
HASH partitioned table	✓	✗
CTE (`SEARCH` and `CYCLE`)	✓	✗
Procedure `OUT` parameters	✓	✗
`CHECK` constraints for foreign tables	✓	✗
Timeout parameter for `pg_terminate_backend`	✓	✗
Auto failover for coordinator	✓	✗
Kubernetes deployment support	✓	✗

1.8.2 Performance-related features

Feature names	SynxDB Elastic	Greenplum
Aggregation pushdown	✓	✗
`CREATE STATISTICS` - `OR` and `IN`/`ANY` statistics	✓	✗
Incremental sort	✓	✗
Incremental sort for window functions	✓	✗
Query pipelining	✓	✗
Query parallelism	✓	✗
Abbrevated keys for sorting	✓	✗
`postgres_fdw` aggregation pushdown	✓	✗
No need to rewrite the whole table when adding a column	✓	✗
Runtime Filter for Join	✓	✗

1.8.3 Security-related features

Feature names	SynxDB Elastic	Greenplum
Transparent Data Encryption (TDE)	✓	✗
Trusted extensions	✓	✗
SCRAM-SHA-256	✓	✗
Encrypted TCP/IP connection when GSSAPI	✓	✗
Row-level security policy	✓	✗

Chapter 2

Deployment

2.1 Deploy SynxDB Elastic

This guide provides a step-by-step walkthrough for installing the SynxDB Elastic, from the initial setup to accessing the console for the first time.

2.1.1 Step 1. Prerequisites

Before beginning the installation, ensure your environment is ready.

Environment and system requirements

Ensure your environment meets the following criteria before proceeding:

- **Operating system:** A Linux server with proper network access.
- **Container runtime:** Docker is installed on your server.
- **Kubernetes cluster:** A running Kubernetes cluster that can:
  - Automatically generate an external hostname/IP for `LoadBalancer` type services.
  - Automatically provision `PersistentVolumes` for `PersistentVolumeClaims`.
- **Object storage:** An S3-compatible object storage service (for example, MinIO) with proper read and write access is available.

- **Client tools:** `kubectl` and `Helm` are installed on your server. If Helm is not installed, follow the official instructions on the Helm website.

Obtain the installation package

The official installation package is required.

- **Action:** Contact technical support to get the installation package.
- **Package size:** About 5.7 GB
- **MD5 checksum:** `fe2e0ea4559bacaee7c3dab39bdb76af`

To ensure the package is complete and not corrupted, verify the checksum after downloading.

Configure Kubernetes CSIDriver

The installation requires a specific setting in the Kubernetes cluster's CSI (Container Storage Interface) Driver.

1. Check the current CSIDriver configuration:

```
kubectl get csidriver
```

2. Edit the CSIDriver. Set `fsGroupPolicy` to `None`.

   Open the driver configuration for editing. For example, if the driver is named `named-disk.csi.cloud-director.vmware.com`, use the following command:

```
kubectl edit csidriver <your-csi-driver-name>
```

3. Update the `spec` section. In the YAML file that opens, find the `spec` section and add or modify the `fsGroupPolicy` line as shown below:

```
spec:
  attachRequired: true
  fsGroupPolicy: None # <-- Adds or modifies this line.
  podInfoOnMount: false
  # ... other settings
```

   Save and close the file to apply the changes.

Import Docker images

The installation package includes several Docker images that need to be loaded and pushed to a container image registry.

1. Load the images. The images are provided as `.tar.gz` files in the `image/` directory of the installation package. Load each one using the `docker load` command. For example:

```
docker load < image/synxdb/synxdb-lakehouse-4.0.0-117013.tar.gz
docker load < image/synxdb/synxdb-elastic-dbaas-1.0-RELEASE-117134.tar.gz
docker load < image/foundationdb/fdb-kubernetes-operator-v2.10.0.tar.gz
docker load < image/foundationdb/fdb-kubernetes-sidecar-7.3.63-1.tar.gz
docker load < image/foundationdb/fdb-kubernetes-monitor-7.3.63.tar.gz
docker load < image/dbeaver/cloudbeaver-23.3.5-884-g156f14-117016-release.tar.gz
docker load < image/minio/minio-RELEASE.2023-10-25T06-33-25Z.tar.gz
docker load < image/minio/mc-RELEASE.2024-11-21T17-21-54Z.tar.gz
```

2. Tag and push the images. After loading, tag the images to match the private registry's URL and then push them. For example:

```
# Tags the image.
docker tag <image_name>:<tag> <your_registry_url>/<repository>/<image_name>:<tag>

# Pushes the image.
docker push <your_registry_url>/<repository>/<image_name>:<tag>
```

   Repeat this for all loaded images.

2.1.2 Step 2. Install dependencies

The service console relies on a few external services. Install these services before proceeding.

Set up S3-compatible object storage

The service console requires an S3-compatible object storage service for metadata backups. Ensure that the service supports the s3v4 authentication protocol.

For testing purposes, you can set up a MinIO instance for metadata backup. To avoid potential conflicts, install it in a dedicated namespace, `minio-metabak`.

1. Install MinIO using the provided Helm chart.

```
helm install minio helm/minio-5.4.0.tgz \
  --namespace minio-metabak \
  --timeout 10m \
  --wait \
  --create-namespace \
  -f example/minio-values.yaml
```

> **Note**
>
> Before running, you might need to edit `example/minio-values.yaml` to point to the MinIO image that you have pushed to the private registry.

Install FoundationDB operator

FoundationDB is used by the service console for its metadata layer.

- **Action:** Install the FoundationDB operator using the Helm chart.
- **Command:**

```
helm install fdb-operator helm/fdb-operator-0.2.0.tgz \
   --namespace fdb \
   --timeout 30m \
   --wait \
   --create-namespace \
   -f example/foundationdb-values.yaml
```

> **Note**
>
> Remember to update `example/foundationdb-values.yaml` with the correct image paths from the registry.

Install CloudBeaver

CloudBeaver provides a web-based SQL client.

- **Action:** Install CloudBeaver using its Helm chart.
- **Command:**

```
helm install cloudbeaver helm/cloudbeaver-0.0.1.tgz \
   --namespace cloudbeaver \
   --timeout 10m \
   --wait \
   --create-namespace \
   -f example/cloudbeaver-values.yaml
```

> **Note**
>
> Update `example/cloudbeaver-values.yaml` with the correct image paths.

Optional: Install monitoring tools

Prometheus and AlertManager are required to enable monitoring and alert features. Follow their official instructions for installation.

2.1.3 Step 3. Install the service console

With the prerequisites and dependencies in place, you can go ahead and install the main service console application.

A note on the database (production vs. testing)

For a production deployment, an external PostgreSQL-compatible database is required for reliability and data persistence. For testing purposes, the service console uses a simpler embedded database by default. This choice is configured in the next step.

Prepare the configuration file

The configuration for the service console is managed in a YAML file. An example is provided at `example/dbaas-values.yaml`.

- **Action:** Open `example/dbaas-values.yaml` and modify it for the environment.
- **Key sections to modify:**
  1. **Images:** Replace all image names with the full paths to the images in the private registry.
  2. **OSS configuration:** This section configures the connection to the S3 object storage. If MinIO was installed in the previous step, the default values should work. The endpoint

will be `http://minio.minio-metabak:32000`.

```
oss:
  moscow:
    vendor: aws
    internal-region: default
    public-region: default
    endpoint: http://minio.minio-metabak:32000 # <-- Check this
    signatureVersion: s3v4
    access-key-id: minio
    access-key-secret: password
```

3. **Database (for production):** Based on the note above, for a production environment, modify the `datasource` section with the external PostgreSQL connection string. Otherwise, the default settings can be used for testing.

4. **Region and profile:** Modify the default region (`moscow`) and deployment profiles to match production requirements. To change the region, globally replace all occurrences of `moscow` with the new name (for example, `ru-central1`).

> **Note**
>
> When you change the region name, you also need to update all other configuration items that refer to the old region name to ensure the settings are consistent.

Run the installation command

Once the configuration file is ready, use Helm to deploy the service console.

- **Action:** Run the `helm install` command.
- **Command:**

```
helm install dbaas-integration helm/dbaas-integration-1.0-RELEASE.tgz \
   --namespace dbaas \
   --timeout 10m \
   --wait \
   --create-namespace \
   -f example/dbaas-values.yaml
```

This command creates a new namespace called `dbaas` and deploys all the necessary components. The `--wait` flag ensures that the command only finishes after the deployment is successful.

2.1.4 Step 4. Access the service console

By default, the service console is not exposed outside the Kubernetes cluster.

Access for testing: port forwarding

For testing, the easiest way to access it is by using port forwarding.

1. Set up port forwarding. Run these commands in the terminal. They will find the correct pod and forward its port to the local machine.

```
# Gets the pod name and saves it to a variable.
export POD_NAME=$(kubectl get pods --namespace dbaas -l "app.kubernetes.io/
name=dbaas-integration,app.kubernetes.io/instance=dbaas-integration" -o jsonpath="
{.items[0].metadata.name}")

# Gets the container port.
export CONTAINER_PORT=$(kubectl get pod --namespace dbaas $POD_NAME -o jsonpath="
{.spec.containers[0].ports[0].containerPort}")

# Starts port forwarding.
echo "Visit http://127.0.0.1:8080 to use your application"
kubectl --namespace dbaas port-forward $POD_NAME 8080:$CONTAINER_PORT
```

2. Open the web console. Keep the port-forwarding command running. You can now access the consoles in a web browser:

    - **User console:** `http://localhost:8080/`
    - **Ops console:** `http://localhost:8080/ops/`

- **Default credentials:**
    - **Username:** `admin`
    - **Password:** `admin`

Access for production: ingress or reverse proxy

For a production environment, set up a Kubernetes Ingress or an HTTP reverse proxy in front of the service console. For security reasons, configure it in the following way:

- **Enable HTTPS:** Expose the service via the HTTPS protocol and redirect all HTTP connections to HTTPS.
- **Restrict console access:** Only expose the user console (`/`) to customers. The ops console (`/ops/`) should only be exposed to the internal network.
- **Redirect for customers:** For customers, configure a redirect from the ops console path to the user console path.

The SynxDB Elastic service console is now installed and accessible.

2.1.5 What's next

After the deployment, see the *quick-start guide* for a quick try of creating necessary resources (such as accounts, users, and warehouses), running SQL queries via the client or console, loading external data, and scaling clusters.

Chapter 3

Load Data

3.1 Load Data Using COPY

`COPY FROM` copies data from a file or standard input in a local file system into a table and appends the data to the table contents. `COPY` is non-parallel: data is loaded in a single process using the SynxDB Elastic coordinator instance. Using `COPY` is only recommended for very small data files.

The `COPY` source file must be accessible to the postgres process on the coordinator host. Specify the `COPY` source file name relative to the data directory on the coordinator host, or specify an absolute path.

SynxDB Elastic copies data from `STDIN` or `STDOUT` using the connection between the client and the coordinator server.

3.1.1 Load from a file

The `COPY` command requests the postgres backend to open the specified file, to read it and append it to the target table. To be able to read the file, the backend needs to have read permissions on the file, and you need to specify the file name using an absolute path on the coordinator host, or using a relative path to the coordinator data directory.

```
COPY <table_name> FROM </path/to/filename>;
```

3.1.2 Load from STDIN

To avoid the problem of copying the data file to the coordinator host before loading the data, `COPY FROM STDIN` uses the Standard Input channel and feeds data directly into the postgres backend. After the `COPY FROM STDIN` command starts, the backend will accept lines of data until a single line only contains a backslash-period (`\.`).

```
COPY <table_name> FROM <STDIN>;
```

3.1.3 Load data using `\copy` in psql

Do not confuse the psql `\copy` command with the `COPY` SQL command. The `\copy` invokes a regular `COPY FROM STDIN` and sends the data from the psql client to the backend. Therefore, any file must locate on the host where the psql client runs, and must be accessible to the user which runs the client.

To avoid the problem of copying the data file to the coordinator host before loading the data, `COPY FROM STDIN` uses the Standard Input channel and feeds data directly into the postgres backend. After the `COPY FROM STDIN` command started, the backend will accept lines of data until a single line only contains a backslash-period (`\.`). psql is wrapping all of this into the handy `\copy` command.

```
\copy <table_name> FROM <filename>;
```

3.1.4 Input format

`COPY FROM` accepts a `FORMAT` parameter, which specifies the format of the input data. The possible values are `TEXT`, `CSV` (Comma Separated Values), and `BINARY`.

```
COPY <table_name> FROM </path/to/filename> WITH (FORMAT csv);
```

The `FORMAT csv` will read comma-separated values. The `FORMAT text` by default uses tabulators to separate the values, the `DELIMITER` option specifies a different character as value delimiter.

```
COPY <table_name> FROM </path/to/filename> WITH (FORMAT text, DELIMITER '|');
```

By default, the default client encoding is used, and you can change this using the `ENCODING` option. This is useful if data is coming from another operating system.

```
COPY <table_name> FROM </path/to/filename> WITH (ENCODING 'latin1');
```

3.2 Load External Data Using Foreign Table

SynxDB Elastic allows you to access data stored in remote data sources using foreign tables. You can use foreign tables to connect to other databases or external data sources (such as CSV files) through a foreign data wrapper (FDW).

3.2.1 Use foreign table

You can create a foreign table using the following command:

> **Note**
>
> Before creating a foreign table, you need to create a foreign server first.

```
CREATE FOREIGN TABLE <external_table_name> (
   col_1 data_type,
   col_2 data_type,
   ...
) SERVER <server_name>
OPTIONS (<option_name> '<option_value>');
```

For example:

```
CREATE FOREIGN TABLE my_foreign_table (
   id INTEGER,
   name TEXT
) SERVER remote_data
OPTIONS (table_name 'external_table');
```

In this example, the table `my_foreign_table` is the foreign table created in the local database, while the actual data is stored in a remote table called `external_table`.

Create foreign table using the LIKE clause

You can use the `LIKE` clause to quickly create a foreign table based on the structure of an existing table. By using this clause, you can define a foreign table without explicitly listing the table's structure.

> **Note**
>
> Foreign tables created using `LIKE` do not inherit the distribution settings of the existing table. These settings should be defined when you create the new foreign table.

The following example shows how to use the `LIKE` clause to create a foreign table:

```
CREATE FOREIGN DATA WRAPPER dummy; -- Creates a foreign data wrapper.
CREATE SERVER s0 FOREIGN DATA WRAPPER dummy; -- Creates a foreign server.
CREATE TABLE ft_source_table(a INT, b INT, c INT); -- Creates a table 'ft_source_table
'.
CREATE FOREIGN TABLE my_foreign_table (LIKE ft_source_table) SERVER s0;  -- Creates a
Foreign Table based on 'ft_source_table'.
\d+ my_foreign_table
                               Foreign table "public.my_foreign_table"
Column |  Type   | Collation | Nullable | Default | FDW options | Storage | Stats
target | Description
--------+---------+-----------+----------+---------+-------------+---------+----------
----+-------------
a      | integer |           |          |         |             | plain   |
   |
b      | integer |           |          |         |             | plain   |
   |
c      | integer |           |          |         |             | plain   |
   |
Server: s0
```

3.2.2 Query a foreign table

After creating a foreign table, you can query it just like a local table.

3.3 Load Data from Kafka Using Kafka FDW

Kafka Foreign Data Wrapper (FDW) enables SynxDB Elastic to connect directly to Apache Kafka, allowing it to read and operate on Kafka data as external tables. This integration allows SynxDB Elastic users to process real-time Kafka data more efficiently, flexibly, and reliably, greatly boosting data processing and business operations.

SynxDB Elastic supports using Kafka FDW to create external tables and import data.

3.3.1 Basic usage

- Create the `kafka_fdw` extension:

```
postgres=# CREATE EXTENSION kafka_fdw;
CREATE EXTENSION
```

- Create an external server and specify Kafka's cluster address:

```
CREATE SERVER kafka_server
FOREIGN DATA WRAPPER kafka_fdw
OPTIONS (mpp_execute 'all segments', brokers 'localhost:9092');
```

- Create user mapping:

```
CREATE USER MAPPING FOR PUBLIC SERVER kafka_server;
```

- Create an external table:

  When creating an external table, you need to specify two metadata columns: `partition` and `offset`, which identify the partition and offset of messages in a Kafka topic. Here is an example:

```
CREATE FOREIGN TABLE kafka_test (
    part int OPTIONS (partition 'true'),
    offs bigint OPTIONS (offset 'true'),
    some_int int,
    some_text text,
    some_date date,
    some_time timestamp
)
```

(continues on next page)

(continued from previous page)

```
SERVER kafka_server OPTIONS
    (format 'csv', topic 'contrib_regress_csv', batch_size '1000', buffer_delay
'1000');
```

Parameter description:

- `batch_size`: The size of data read from Kafka at once.
- `buffer_delay`: The timeout for getting data from Kafka.

3.3.2 Supported data formats

Currently, `CSV` and `JSON` data formats are supported.

3.3.3 Query

You can specify the message partition and offset in your query by using the `partition` or `offset` column condition:

```
SELECT * FROM kafka_test WHERE part = 0 AND offs > 1000 LIMIT 60;
```

You can also specify multiple conditions:

```
SELECT * FROM kafka_test WHERE (part = 0 AND offs > 100) OR (part = 1 AND offs > 300)
OR (part = 3 AND offs > 700);
```

3.3.4 Message producer

Currently, Kafka FDW supports inserting data into external tables, which acts as a message producer for Kafka. You only need to use the `INSERT` statement.

```
INSERT INTO kafka_test(part, some_int, some_text)
VALUES
    (0, 5464565, 'some text goes into partition 0'),
    (1, 5464565, 'some text goes into partition 1'),
    (0, 5464565, 'some text goes into partition 0'),
    (3, 5464565, 'some text goes into partition 3'),
    (NULL, 5464565, 'some text goes into partition selected by kafka');
```

When inserting data, you can specify `partition` to specify which partition to insert into.

3.3.5 Data import

To use Kafka FDW for data import, you can create custom functions, such as the `INSERT INTO SELECT` statement. The basic principle is to fetch all data from the external table and insert it into the target table sequentially.

Here is a simple example, which you can modify according to your needs:

```
CREATE OR REPLACE FUNCTION import_kafka_data(
  src_table_name text,
  dest_table_name text,
  dest_table_columns text[]
) RETURNS void AS $$

DECLARE
    current_row RECORD;
    current_part integer;
    current_offs bigint;
    max_off bigint;
    import_progress_table_name text;
    max_off_result bigint;
BEGIN

    import_progress_table_name := src_table_name || '_import_progress';

    -- Creates progress record table.
    EXECUTE FORMAT('CREATE TABLE IF NOT EXISTS %I (part integer, offs bigint NOT NULL)
', import_progress_table_name);

    -- The number of partitions in the topic table might change, so reinitialize
before each import.
    EXECUTE FORMAT('INSERT INTO %I SELECT DISTINCT part, 0 FROM %I', import_progress_
table_name, src_table_name);

    -- Imports data partition by partition.
    FOR current_row IN
        EXECUTE FORMAT('SELECT part, offs FROM %I', import_progress_table_name)
    LOOP
        current_part := current_row.part;
        current_offs := current_row.offs;
```

(continues on next page)

(continued from previous page)

```
        -- Gets the maximum offset for the current partition.
        EXECUTE FORMAT('SELECT MAX(offs) FROM %I WHERE part = %s', src_table_name,
current_part) INTO max_off_result;
        max_off := max_off_result;

        -- Skips if there is no new data.
        IF max_off+1 = current_offs THEN
            CONTINUE;
        END IF;

        -- Imports data.
        EXECUTE FORMAT('
            INSERT INTO %I (%s)
            SELECT %s
            FROM %I
            WHERE part = %s AND offs >= %s AND offs <= %s',
            dest_table_name,
            array_to_string(dest_table_columns, ', '),
            array_to_string(dest_table_columns, ', '),
            src_table_name,
            current_part,
            current_offs,
            max_off
        );

        -- Updates import progress.
        EXECUTE FORMAT('UPDATE %I SET offs = %s WHERE part = %s', import_progress_
table_name, max_off + 1, current_part);
    END LOOP;

    RETURN;
END;
$$ LANGUAGE plpgsql;
```

When executing the query, call this function, passing in the external table name, target table name, and the fields to be imported. Here is an example:

```
SELECT import_kafka_data('kafka_test', 'dest_table_fdw', ARRAY['some_int', 'some_text
', 'some_date', 'some_time']);
```

Scheduled import

To create a scheduled task to import data in the background, you can use the Task feature in SynxDB Elastic to execute the import function periodically.

```
CREATE TASK import_kafka_data schedule '1 seconds' AS $$SELECT import_kafka_data(
'kafka_test', 'dest_table_fdw', ARRAY['some_int', 'some_text', 'some_date', 'some_time
']);$$;
```

In the example above, the function to import data is called every second. This setup allows for the continuous use of Kafka FDW to import data from the source external table into the target table.

3.4 Load Data from Object Storage and HDFS

You can use the `datalake_fdw` extension to load data from an object storage (such as Amazon S3 and other major cloud providers), HDFS, and ORC tables in Hive into SynxDB Elastic for data query and access.

To install the `datalake_fdw` extension to the database, execute the SQL statement `CREATE EXTENSION data_fdw;`.

```
CREATE EXTENSION datalake_fdw;
```

Currently, supported data formats are CSV, TEXT, ORC, and PARQUET.

> **Note**
>
> `datalake_fdw` does not support loading data in parallel.

For information on how to load tables from Hive into SynxDB Elastic, see *Load Data from Hive Data Warehouse*.

3.4.1 Load data from object storage

You can load data from major cloud providers like Amazon S3, Google Cloud Storage, and Microsoft Azure Blob Storage into SynxDB Elastic. Follow these steps:

1. Create a foreign table wrapper `FOREIGN DATA WRAPPER`. Note that there are no options in the SQL statement below, and you need to execute it exactly as provided.

   ```
   CREATE FOREIGN DATA WRAPPER datalake_fdw
   HANDLER datalake_fdw_handler
   VALIDATOR datalake_fdw_validator
   OPTIONS ( mpp_execute 'all segments' );
   ```

2. Create an external server `foreign_server`.

   ```
   CREATE SERVER foreign_server
   FOREIGN DATA WRAPPER datalake_fdw
   OPTIONS (host 'xxx', protocol 's3', isvirtual 'false', ishttps 'false');
   ```

The options in the above SQL statement are explained as follows:

Option name	Description	Details
`host`	Sets the host information for accessing the object storage.	Required: Must be set Example: • Host for private cloud: `192.168.1.1:9000`
`protocol`	Specifies the cloud platform for the object storage.	Required: Must be set Options: • `s3`: Amazon S3 and S3-compatible storage (uses v4 signature)
`isvirtual`	Use **virtual-host-style** or **path-host-style** to parse the host of the object storage.	Required: Optional Options: • `true`: Uses **virtual-host-style**. • `false`: Uses **path-host-style**. Default value: `false`
`ishttps`	Whether to use HTTPS to access the object storage.	Required: Optional Options: • `true`: Uses HTTPS. • `false`: Does not use HTTPS. Default value: `false`

3. Create a user mapping.

```
CREATE USER MAPPING FOR gpadmin SERVER foreign_server
OPTIONS (user 'gpadmin', accesskey 'xxx', secretkey 'xxx');
```

The options in the above SQL statement are explained as follows:

Option name	Description	Required
`user`	Creates the specific user specified by `foreign_server`.	Yes
`accesskey`	The key needed to access the object storage.	Yes
`secretkey`	The secret key needed to access the object storage.	Yes

4. Create a foreign table `example`. After creating it, the data on the object storage is loaded into SynxDB Elastic, and you can query this table.

```
CREATE FOREIGN TABLE example(
a text,
b text
)
SERVER foreign_server
```

(continues on next page)

(continued from previous page)

```
OPTIONS (filePath '/test/parquet/', compression 'none', enableCache 'false',
format 'parquet');
```

The options in the SQL statement above are explained as follows:

Option name	Description	Details

`filePath`	Sets the specific path for the target foreign table.	• Required: Must be set
		• Path format should be `/bucket/prefix`.
		• Example:
		  ◦ If the bucket name is `test-bucket` and the path is `bucket/test/orc_file_folder/`, and there are files like `0000_0`, `0001_1`, `0002_2`, then to access file `0000_0`, set `filePath` to `filePath '/test-bucket/test/orc_file_folder/0000_0'`.
		  ◦ To access all files in `test/orc_file_folder/`, set `filePath` to `filePath '/test-bucket/test/orc_file_folder/'`.
		• Note: `filePath` is parsed in the format `/bucket/prefix/`. Incorrect formats might lead to errors, such as:
		  ◦ `filePath 'test-bucket/test/orc_file_folder/'`
		  ◦ `filePath '/test-bucket/test/orc_file_folder/0000_0'`

`compression`	Sets the write compression format. Currently supports snappy, gzip, zstd, lz4.	• Required: Optional
		• Options:
		  ◦ `none`: Supports CSV, ORC, TEXT, PARQUET.
		  ◦ `gzip`: Supports CSV, TEXT, PARQUET.
		  ◦ `snappy`: Supports PARQUET.
		  ◦ `zstd`: Supports PARQUET.
		  ◦ `lz4`: Supports PARQUET.
		• Default value: `none`, which means no compression. Not setting this value means no compression.

continues on next page

Table 3 – continued from previous page

Option name	Description	Details

`enableCache`	Specifies whether to use Gopher caching.	• Required: Optional
		• Options:
		  ○ `true`: Enables Gopher caching.
		  ○ `false`: Disables Gopher caching.
		• Default value: `false`
		• Deleting the foreign table does not automatically clear its cache. To clear the cache, you need to manually run a specific SQL function, such as: `select gp_toolkit._gopher_cache_free_relation_name(text);`

`format`	The file format supported by FDW.	• Required: Must be set
		• Options:
		  ○ `csv`: Read, Write
		  ○ `text`: Read, Write
		  ○ `orc`: Read, Write
		  ○ `parquet`: Read, Write

5. Use `insert` and `select` statements to add data to and query the data from the foreign table `example` like a normal table.

```
insert into example values ('1', '2');

select * from example;
```

3.4.2 Load Iceberg table data from S3 (without an external metadata service)

This section describes how to configure SynxDB Elastic to directly load Apache Iceberg tables stored on Amazon S3 or other compatible object storage without depending on an external metadata catalog (such as Hive Metastore or a REST Catalog).

This feature is primarily intended for quick, read-only querying and analysis of existing Iceberg data.

Prerequisites: Configure the S3 connection file

To enable this feature, you need to create a configuration file named `s3.conf` on all coordinator nodes of the SynxDB Elastic cluster. This file provides the underlying `datalake_agent` with the necessary connection and authentication information to access S3.

1. Get the Namespace. First, you need to determine the Kubernetes namespace where the SynxDB Elastic cluster is located.

```
kubectl get ns
```

2. Edit the `connector-config` ConfigMap. Use the `kubectl edit` command to edit `connector-config`. Replace `<namespace>` in the command with the actual namespace obtained in the previous step.

```
kubectl edit cm connector-config -n <namespace>
```

3. Add the content of `s3.conf` to the `ConfigMap`. After running the command, a text editor (like `vi`) will open. In the `data:` section, following the format of `gphdfs.conf`, add a new key named `s3.conf` and paste the entire content of `s3.conf` as its value. For example:

```
apiVersion: v1
kind: ConfigMap
metadata:
name: connector-config
namespace: <namespace>
data:
gphdfs.conf: |
   hdfs-cluster-1:
      hdfs_namenode_host: <namenode_ip>
      ...
# --- Add the key and value for s3.conf here ---
s3.conf: |
   s3_cluster:
      fs.s3a.endpoint: http://127.0.0.1:8000
      fs.s3a.access.key: admin
      fs.s3a.secret.key: password
      fs.s3a.path.style.access: true
      fs.defaultFS: s3a://
      fs.s3a.impl: org.apache.hadoop.fs.s3a.S3AFileSystem
```

4. Save and exit. Save the file and close the editor. Kubernetes will automatically update the `ConfigMap` and mount the new `s3.conf` file into the corresponding Pods.

Procedures

1. Create a foreign data wrapper. You can skip this step if it already exists.

```
CREATE FOREIGN DATA WRAPPER datalake_fdw
HANDLER datalake_fdw_handler
VALIDATOR datalake_fdw_validator
OPTIONS (mpp_execute 'all segments');
```

2. Create a foreign server pointing to the S3 service. This is a standard S3 server definition.

```
CREATE SERVER s3_server
FOREIGN DATA WRAPPER datalake_fdw
OPTIONS (host 'your_s3_host', protocol 's3');
```

   - `host`: Specifies the host information for accessing the object storage.
   - `protocol`: For S3 or compatible storage, set this to `s3`.

3. Create a foreign table to map to the Iceberg data on S3.

```
CREATE FOREIGN TABLE iceberg_s3_table (
   -- Define the table columns here, which must match the Iceberg table's schema.
   id int,
   name text,
   create_date date  -- If it is a partitioned table, the partition key must also
be defined as a column.
)
SERVER s3_server
OPTIONS (
   format 'iceberg',
   catalog_type 's3',
   server_name 's3_cluster',
   filePath '/your_bucket/path/to/warehouse/',
   table_identifier 'your_db.your_table'
);
```

   - `format`: Specifies the file format. For this scenario, it is fixed to `'iceberg'`.

- `catalog_type`: Specifies the catalog type. For S3 scenarios without a catalog, it is fixed to `'s3'`.
- `server_name`: Specifies the name of the cluster configuration defined in the `s3.conf` file. In this example, it is `'s3_cluster'`.
- `filePath`: Points to the root path of the Iceberg "warehouse" or the parent directory of the database. The format is `/bucket_name/prefix/`.
- `table_identifier`: Specifies the identifier of the table to be accessed, in the format `<database_name>.<table_name>`. SynxDB Elastic concatenates this identifier with `filePath` to locate the final table data path.

Examples

Example 1: Query a non-partitioned table. Assume the path to the Iceberg table on S3 is `s3a://ossext-ci-test/warehouse/iceberg/warehouse/default/simple_table`.

1. Create the foreign table `iceberg_simple`:

```
CREATE FOREIGN TABLE iceberg_simple (
   id int,
   name text
)
SERVER s3_server
OPTIONS (
   filePath '/ossext-ci-test/warehouse/iceberg/warehouse/',
   catalog_type 's3',
   server_name 's3_cluster',
   table_identifier 'default.simple_table',
   format 'iceberg'
);
```

2. Query the data:

```
SELECT * FROM iceberg_simple WHERE id = 1;
```

Example 2: Query a partitioned table. Assume the Iceberg table `partitioned_table` on S3 is partitioned by the `create_date` field, and its path is `s3a://ossext-ci-test/warehouse/iceberg/warehouse/testdb/partitioned_table`.

1. Create the foreign table `iceberg_partitioned`. Note that the partition key `create_date` must be included in the column definitions.

```
CREATE FOREIGN TABLE iceberg_partitioned (
   id int,
   name text,
   age int,
   department text,
   create_date date
)
SERVER s3_server
OPTIONS (
   filePath '/ossext-ci-test/warehouse/iceberg/warehouse/',
   catalog_type 's3',
   server_name 's3_cluster',
   table_identifier 'testdb.partitioned_table',
   format 'iceberg'
);
```

2. Query the data:

```
SELECT name, department FROM iceberg_partitioned WHERE create_date = '2025-05-20';
```

Limitations and notes

- Read-only operations: Iceberg foreign tables created using this method only support `SELECT` queries. Write operations such as `INSERT`, `UPDATE`, and `DELETE` are not supported.
- Authentication method: This feature only uses the `s3.conf` configuration file for authentication. The `CREATE USER MAPPING` method described in the documentation is not applicable to this scenario.
- Path concatenation: Ensure that `filePath` and `table_identifier` are set correctly. The system locates the table data using the logic `filePath + table_identifier`. `filePath` should typically point to the warehouse root directory that contains multiple database directories.

3.4.3 Read Iceberg tables on S3 via Polaris Catalog

This section explains how to query Apache Iceberg tables stored on Amazon S3 or other compatible object storage in SynxDB Elastic by connecting to a Polaris Catalog service.

This feature allows you to use an external, centralized metadata service to manage Iceberg tables while using the powerful query capabilities of SynxDB Elastic for data analysis. Iceberg foreign tables created with this method currently only support `SELECT` queries; write operations like `INSERT`, `UPDATE`, and `DELETE` are not supported.

Core concepts

Unlike accessing the filesystem directly, accessing Iceberg tables via a catalog service requires SynxDB Elastic to communicate with two separate external systems:

- Polaris Catalog Service: A service for storing and managing Iceberg table metadata (such as schema, partition information, and snapshots).
- S3 Object Storage Service: An external service for storing the actual data files (for example, parquet files).

Therefore, you need to create two independent sets of `SERVER` and `USER MAPPING` objects to configure and authenticate the connections for these two services respectively.

Prerequisites

- Network connectivity:
    - Ensure that the SynxDB Elastic cluster has network access to the `host` address of the external S3 service. This may require configuring appropriate firewall outbound rules or network policies. The requirements for accessing S3 are the same as for standard S3 foreign tables.
    - Ensure that the Polaris Catalog service can access the SynxDB Elastic cluster.
- Credentials:
    - Prepare the authentication credentials (`accesskey` and `secretkey`) required to access the S3 service.
    - Prepare the OAuth2 authentication credentials (`client_id` and `client_secret`) required to access the Polaris Catalog service.

Procedure to read Iceberg tables on S3

1. Create the FOREIGN DATA WRAPPER `datalake_fdw`. You can skip this step if it already exists.

```
CREATE EXTENSION IF NOT EXISTS datalake_fdw;

CREATE FOREIGN DATA WRAPPER datalake_fdw
HANDLER datalake_fdw_handler
VALIDATOR datalake_fdw_validator
OPTIONS (mpp_execute 'all segments');
```

2. Configure the connection and authentication for the S3 service. Create a `SERVER` object and a corresponding `USER MAPPING` for the external S3 service.

```
-- 1. Create a server object for the S3 service.
CREATE SERVER s3_data_server
FOREIGN DATA WRAPPER datalake_fdw
OPTIONS (host 'your_s3_host:port', protocol 's3', ishttps 'false');

-- 2. Create a user mapping for the S3 server to provide authentication
credentials.
CREATE USER MAPPING FOR gpadmin
SERVER s3_data_server
OPTIONS (user 'gpadmin', accesskey 'YOUR_S3_ACCESS_KEY', secretkey 'YOUR_S3_
SECRET_KEY');
```

3. Configure the connection and authentication for the Polaris Catalog service. Similarly, create a dedicated `SERVER` object and `USER MAPPING` for the internal Polaris Catalog service.

```
-- 1. Create a server object for the Polaris Catalog service.
CREATE SERVER polaris_catalog_server
FOREIGN DATA WRAPPER datalake_fdw
OPTIONS (polaris_server_url 'http://polaris-service-url:8181/api/catalog');

-- 2. Create a user mapping for the Polaris server to provide OAuth2
authentication credentials.
CREATE USER MAPPING FOR gpadmin
SERVER polaris_catalog_server
OPTIONS (client_id 'your_client_id', client_secret 'your_client_secret', scope
'PRINCIPAL_ROLE:ALL');
```

4. Create a foreign table to map to the Iceberg table managed by the Polaris Catalog.

```
CREATE FOREIGN TABLE my_iceberg_table (
   name text,
   score decimal(16, 2)
)
SERVER s3_data_server -- Note: The SERVER here points to the S3 data server.
OPTIONS (
   format 'iceberg',
   catalog_type 'polaris',
   table_identifier 'polaris.testdb.mytable',
   server_name 'polaris_catalog_server', -- [Key] Specifies which server to get
metadata from.
   filePath '/your-bucket/warehouse/polaris' -- [Key] Still need to specify the
data root path on S3.
);
```

`OPTIONS` parameter details:

- `format`: Specifies the file format. For this scenario, it is fixed to `'iceberg'`.
- `catalog_type`: Specifies the catalog type, fixed to `'polaris'`.
- `table_identifier`: The full identifier of the table in the Polaris Catalog, in the format `<catalog_name>.<db_name>.<table_name>`.
- `server_name`: [Key] Specifies the name of the Polaris Catalog server used for fetching metadata, which is `polaris_catalog_server` created in Step 3.
- `filePath`: [Key] The root or warehouse path on S3 where the Iceberg data files are stored. This parameter is still required.

Complete example

```
-- Step 1: Create FDW.
CREATE EXTENSION IF NOT EXISTS datalake_fdw;
CREATE FOREIGN DATA WRAPPER datalake_fdw HANDLER datalake_fdw_handler VALIDATOR
datalake_fdw_validator OPTIONS ( mpp_execute 'all segments' );

-- Step 2: Configure S3 access.
CREATE SERVER s3_server FOREIGN DATA WRAPPER datalake_fdw OPTIONS (host '192.168.50.
102:8002', protocol 's3', ishttps 'false');
```

(continues on next page)

(continued from previous page)

```
CREATE USER MAPPING FOR gpadmin SERVER s3_server OPTIONS (user 'gpadmin', accesskey
'0QpV601CpxpfUaVmQm1Y', secretkey 'daRYWISTvibNnnxCqS8MEgOGZWpFHtL2EkDD5YRv');

-- Step 3: Configure Polaris Catalog access.
CREATE SERVER polaris_server FOREIGN DATA WRAPPER datalake_fdw OPTIONS (polaris_
server_url 'http://192.168.50.102:8181/api/catalog');
CREATE USER MAPPING FOR gpadmin SERVER polaris_server OPTIONS (client_id 'root',
client_secret 'secret', scope 'PRINCIPAL_ROLE:ALL');

-- Step 4: Create foreign table and query.
CREATE FOREIGN TABLE iceberg_rest_table (
   name text,
   score decimal(16,2)
)
SERVER s3_server
OPTIONS (
   filePath '/your-actual-bucket/warehouse/polaris',
   catalog_type 'polaris',
   table_identifier 'polaris.testdb1.table27',
   server_name 'polaris_server',
   format 'iceberg'
);

-- Query data
SELECT * FROM iceberg_rest_table LIMIT 10;
```

3.4.4 Load data from HDFS without authentication

You can load data from HDFS into SynxDB Elastic. The following sections explain how to load data from an HDFS cluster without authentication. SynxDB Elastic also supports loading data from an HDFS HA (High Availability) cluster, which is also explained below.

Load data from HDFS in the simple mode, which is the basic HDFS mode without using complex security authentication. For details, see the Hadoop documentation: Hadoop in Secure Mode. The steps are as follows:

1. Create an external table wrapper `FOREIGN DATA WRAPPER`. Note that there are no options in the SQL statement below, and you need to execute the statement exactly as provided.

```
CREATE FOREIGN DATA WRAPPER datalake_fdw
HANDLER datalake_fdw_handler
VALIDATOR datalake_fdw_validator
OPTIONS ( mpp_execute 'all segments' );
```

2. Create an external server. In this step, you can create an external server for a single-node HDFS or for HA (High Availability) HDFS.

   - Create an external server `foreign_server` for a single-node HDFS:

   ```
   CREATE SERVER foreign_server FOREIGN DATA WRAPPER datalake_fdw
   OPTIONS (
       protocol 'hdfs',
       hdfs_namenodes 'xx.xx.xx.xx',
       hdfs_port '9000',
       hdfs_auth_method 'simple',
       hadoop_rpc_protection 'authentication');
   ```

   The options in the above SQL statement are explained as follows:

Option name	Description	Details
`protocol`	Specifies the Hadoop platform.	○ Required: Must be set ○ Setting: Fixed as `hdfs`, which means Hadoop platform, cannot be changed. ○ Default value: `hdfs`
`hdfs_namenodes`	Specifies the namenode host for accessing HDFS.	○ Required: Must be set ○ Example: For example, `hdfs_namenodes '192.168.178.95:9000'`
`hdfs_auth_method`	Specifies the authentication mode for accessing HDFS.	○ Required: Must be set ○ Options: * `simple`: Uses Simple authentication to access HDFS. ○ Note: To access in Simple mode, set the value to `simple`, for example, `hdfs_auth_method 'simple'`.

continues on next page

Table 4 – continued from previous page

Option name	Description	Details
`hadoop_rpc_protection`	Configures the authentication mechanism for setting up a SASL connection.	◦ Required: Must be set ◦ Options: Three values are available: `authentication`, `integrity`, and `privacy`. ◦ Note: This option must match the `hadoop.rpc.protection` setting in the HDFS configuration file `core-site.xml`. For more details, see the Hadoop documentation Explanation of core-site.xml.

- Create an external server for a multi-node HA cluster. The HA cluster supports node failover. For more information about HDFS high availability, see the Hadoop documentation HDFS High Availability Using the Quorum Journal Manager.

  To load an HDFS HA cluster, you can create an external server using the following template:

```
CREATE SERVER foreign_server
        FOREIGN DATA WRAPPER datalake_fdw
        OPTIONS (
        protocol 'hdfs',
        hdfs_namenodes 'mycluster',
        hdfs_auth_method 'simple',
        hadoop_rpc_protection 'authentication',
        is_ha_supported 'true',
        dfs_nameservices 'mycluster',
        dfs_ha_namenodes 'nn1,nn2,nn3',
        dfs_namenode_rpc_address '192.168.178.95:9000,192.168.178.160:9000,
192.168.178.186:9000',
        dfs_client_failover_proxy_provider 'org.apache.hadoop.hdfs.server.
namenode.ha.ConfiguredFailoverProxyProvider');
```

  In the above SQL statement, `protocol`, `hdfs_namenodes`, `hdfs_auth_method`, and `hadoop_rpc_protection` are the same as in the single-node example. The HA-specific options are explained as follows:

Option name	Description	Details
`is_ha_supported`	Specifies whether to access the HDFS HA service (high availability).	○ Required: Must be set ○ Setting: Set to `true`. ○ Default value: /
`dfs_nameservices`	When `is_ha_supported` is `true`, specify the name of the HDFS HA service to access.	○ Required: If using an HDFS HA cluster, must be set. ○ Matches the `dfs.ha.namenodes.mycluster` item in the HDFS config file `hdfs-site.xml`. ○ Note: For example, if `dfs.ha.namenodes.mycluster` is `cluster`, set this option as `dfs_nameservices 'mycluster'`.
`dfs_ha_namenodes`	When `is_ha_supported` is `true`, specify the accessible nodes for HDFS HA.	○ Required: If using an HDFS HA cluster, must be set. ○ Setting: Matches the value of the `dfs.ha.namenodes.mycluster` item in the HDFS config file `hdfs-site.xml`. ○ Note: For example, `dfs_ha_namenodes 'nn1,nn2,nn3'`.
`dfs_namenode_rpc_address`	When `is_ha_supported` is `true`, specifies the IP addresses of the high availability nodes in HDFS HA.	○ Required: If using an HDFS HA cluster, must be set. ○ Setting: Refer to the `dfs.ha_namenodes` configuration in the HDFS `hdfs-site.xml` file. The node address matches the `namenode` address in the configuration. ○ Note: For example, if `dfs.ha.namenodes.mycluster` has three namenodes named `nn1`, `nn2`, `nn3`, find their addresses from the HDFS configuration file and enter them into this field. `dfs_namenode_rpc_address '192.168.178.95:9000,192.168.178.160:9000,192.168.178.186:9000'`
`dfs_client_failover_pro`	Specifies whether HDFS HA has failover enabled.	○ Required: If using an HDFS HA cluster, must be set. ○ Setting: Set to the default value: `dfs_client_failover_proxy_provider 'org.apache.hadoop.hdfs.server.namenode.ha.Configur` ○ Default value: /

3. Create a user mapping.

```
CREATE USER MAPPING FOR gpadmin SERVER foreign_server
OPTIONS (user 'gpadmin');
```

In the above statement, the `user` option specifies the specific user for `foreign_server` and must be set.

4. Create the foreign table `example`. After creating it, the data from object storage is already loaded into SynxDB Elastic, and you can query this table.

```
CREATE FOREIGN TABLE example(
a text,
b text
)
SERVER foreign_server
OPTIONS (filePath '/test/parquet/', compression 'none', enableCache 'false',
format 'parquet');
```

The options in the above SQL statement are explained as follows:

Option name	Description	Details

`filePath`	Sets the specific path of the target foreign table.	• Required: Must be set
		• Setting: The path format should be `/bucket/prefix`. Example: If the bucket name is `test-bucket` and the path is `bucket/test/orc_file_folder/`, and there are multiple files like `0000_0`, `0001_1`, `0002_2` in that path, you can access the `0000_0` file by setting `filePath '/test-bucket/test/orc_file_folder/0000_0'`. To access all files in `test/orc_file_folder/`, set `filePath '/test-bucket/test/orc_file_folder/'`.
		• Note: `filePath` should follow the `/bucket/prefix/` format. Incorrect formats might lead to errors, such as:
		  ○ `filePath 'test-bucket/test/orc_file_folder/'`
		  ○ `filePath '/test-bucket/test/orc_file_folder/0000_0'`

continues on next page

Table 6 – continued from previous page

Option name	Description	Details

`compression`	Sets the compression format for writing. Currently supports snappy, gzip, zstd, lz4 formats.	• Required: Optional
		• Setting:
		  ◦ `none`: Supports CSV, ORC, TEXT, PARQUET formats.
		  ◦ `gzip`: Supports CSV, TEXT, PARQUET formats.
		  ◦ `snappy`: Supports PARQUET formats.
		  ◦ `zstd`: Supports PARQUET format.
		  ◦ `lz4`: Supports PARQUET format.
		• Default value: `none`, which means no compression. Not setting this value also means no compression.

`enableCache`	Specifies whether to use the Gopher cache.	• Required: Optional
		• Setting:
		  ◦ `true`: Enables Gopher cache.
		  ◦ `false`: Disables Gopher cache.
		• Default: `false`
		• Note: Deleting a foreign table does not automatically clear the cache. To clear the cache for this table, you need to manually run a specific SQL function, for example:
		`select gp_toolkit._gopher_cache_free_relation_name(text);`

`format`	The file format supported by FDW.	• Required: Must be set
		• Setting:
		  ◦ `csv`: Readable, writable
		  ◦ `text`: Readable, writable
		  ◦ `orc`: Readable, writable
		  ◦ `parquet`: Readable, writable

3.4.5 Load data from HDFS using Kerberos authentication

This section provides instructions for establishing secure data integration between SynxDB Elastic and HDFS using Kerberos authentication. The document covers the following integration scenarios:

1. Reading CSV files from HDFS with Kerberos authentication
2. Writing data to CSV files in HDFS with Kerberos authentication
3. Reading Apache Iceberg format files from HDFS with Kerberos authentication

Prerequisites

The following configurations must be completed in your SynxDB Elastic cluster:

- Configure `hdfs.keytab`
- Configure `krb5.conf`
- Configure `coredns`
- Configure `gphdfs.conf` (required only for Iceberg format configuration)
- Complete Kerberos Authentication

Step 1: Prepare required files from Hadoop cluster

On the Hadoop cluster, locate and copy the following files:

```
# Locates the files
ls /opt/hadoop-3.1.4/etc/hadoop/hdfs.keytab
ls /etc/krb5.conf

# Copies the files to the SynxDB cluster
scp /opt/hadoop-3.1.4/etc/hadoop/hdfs.keytab root@<synxdb_ip>:~/
scp /etc/krb5.conf root@<synxdb_ip>:~/
```

Step 2: Configure SynxDB Elastic cluster

Perform the following configurations on your SynxDB Elastic cluster:

1. Configure hdfs.keytab

```
# On SynxDB cluster, retrieve the namespace information
kubectl get ns

# Updates the kerberos-keytab secret with the hdfs.keytab file
kubectl -n <namespace> get secret kerberos-keytab -o json | \
jq --arg new_value "$(base64 -i hdfs.keytab)" '.data["hdfs.keytab"] = $new_value' | \
kubectl -n <namespace> apply -f -
```

2. Configure krb5.conf

```
# On SynxDB cluster, access the kerberos configuration
kubectl edit cm kerberos-config -n <namespace>

# Implements the following configuration
[logging]
 default = FILE:/var/log/krb5libs.log
 kdc = FILE:/var/log/krb5kdc.log
 admin_server = FILE:/var/log/kadmind.log

[libdefaults]
 dns_lookup_realm = false
 ticket_lifetime = 24h
 renew_lifetime = 7d
 forwardable = true
 rdns = false
 pkinit_anchors = FILE:/etc/pki/tls/certs/ca-bundle.crt
 default_realm = EXAMPLE.COM

[realms]
 EXAMPLE.COM = {
  kdc = <kdc_ip>
  admin_server = <admin_server_ip>
 }

[domain_realm]
 .example.com = EXAMPLE.COM
 example.com = EXAMPLE.COM
```

3. Configure CoreDNS

```
# On SynxDB cluster, access the CoreDNS configuration
kubectl -n kube-system edit cm coredns

# Implements the Hadoop cluster IP address and domain name mapping
data:
  Corefile: |
    .:53 {
        errors
```

(continues on next page)

(continued from previous page)

```
        health {
           lameduck 5s
        }
        ready
        hosts {
            <hadoop_ip> <hadoop_hostname>
            fallthrough
        }
        kubernetes cluster.local in-addr.arpa ip6.arpa {
            pods insecure
            fallthrough in-addr.arpa ip6.arpa
        }
    }

# Restart the CoreDNS service
kubectl -n kube-system scale deploy coredns --replicas=0
kubectl -n kube-system scale deploy coredns --replicas=2
```

4. Configure gphdfs.conf

```
# On SynxDB cluster, access the connector configuration
kubectl edit cm connector-config -n <namespace>

# Implement the following configuration
hdfs-cluster-1:
    hdfs_namenode_host: <namenode_ip>
    hdfs_namenode_port: 9000
    hdfs_auth_method: kerberos
    krb_principal: hdfs/<namenode_ip>@EXAMPLE.COM
    krb_principal_keytab: /etc/kerberos/keytab/hdfs.keytab
    krb_service_principal: hdfs/<namenode_ip>@EXAMPLE.COM
    hadoop_rpc_protection: authentication
    data_transfer_protection: privacy
    data_transfer_protocol: true
```

Attention

The default port for the `data_lake` agent has been changed from `5888` to `3888` to avoid

conflict with PXF.

5. Complete Kerberos authentication

```
# On SynxDB cluster, access the namespace
kubectl exec -it cloudberry-proxy-0 -n <namespace> -- bash

# Installs the required Kerberos client tools
sudo su
yum -y install krb5-libs krb5-workstation
exit

# Initializes the Kerberos ticket
kinit -k -t /etc/kerberos/keytab/hdfs.keytab hdfs/<namenode_ip>@EXAMPLE.COM

# Verifies the Kerberos ticket
klist
```

Read and Write CSV Files

Step 1: Prepare data in HDFS

On the Hadoop cluster, create and verify the CSV data:

```
# Creates sample CSV data
hdfs dfs -cat /tmp/hdfs_hd_csv/*
1,lightning
2,cloudberry
3,synxml
```

Step 2: Configure SynxDB Elastic for CSV access

On the SynxDB Elastic cluster, configure the external table:

```
-- Initializes the foreign data wrapper extension
CREATE EXTENSION datalake_fdw;

-- Configures the HDFS foreign data wrapper
CREATE FOREIGN DATA WRAPPER hdfs_fdw
    HANDLER datalake_fdw_handler
```

(continues on next page)

(continued from previous page)

```
    VALIDATOR datalake_fdw_validator
    OPTIONS (mpp_execute 'all segments');

-- Establishes the HDFS server connection
CREATE SERVER hdfs_server FOREIGN DATA WRAPPER hdfs_fdw
    OPTIONS (
        Protocol 'hdfs',
        hdfs_namenodes '<namenode_ip>',
        hdfs_port '9000',
        hdfs_auth_method 'kerberos',
        krb_principal 'hdfs/<namenode_ip>@EXAMPLE.COM',
        krb_principal_keytab '/etc/kerberos/keytab/hdfs.keytab',
        hadoop_rpc_protection 'authentication',
        data_transfer_protocol 'true'
    );

-- Configures user mapping
CREATE USER MAPPING FOR gpadmin SERVER hdfs_server
    OPTIONS (user 'gpadmin');

-- Creates the external table definition
CREATE FOREIGN TABLE ext_t_hdfs(
    a int,
    b text
)
SERVER hdfs_server
OPTIONS (
    filePath '/tmp/hdfs_hd_csv',
    compression 'none',
    enableCache 'false',
    format 'csv',
    delimiter ',',
    NULL E'\\N'
);
```

Step 3: Read and write data

On the SynxDB Elastic cluster, perform data operations:

```
-- Execute a data retrieval query
SELECT * FROM ext_t_hdfs;
 a |     b
---+-----------
 1 | lightning
 2 | cloudberry
 3 | synxml
(3 rows)

-- Perform data insertion
INSERT INTO ext_t_hdfs VALUES
(4, 'enterprise'),
(5, 'public cloud');

-- Verify the data operation
SELECT * FROM ext_t_hdfs;
 a |      b
---+--------------
 1 | lightning
 2 | cloudberry
 3 | synxml
 5 | public cloud
 4 | enterprise
(5 rows)
```

Step 4: Verify data in HDFS

On the Hadoop cluster, verify the written data:

```
# Verify the data
hdfs dfs -ls /tmp/hdfs_hd_csv/
hdfs dfs -cat /tmp/hdfs_hd_csv/*
```

Read Iceberg files

Step 1: Create Iceberg table in HDFS

On the Hadoop cluster, create and populate the Iceberg table:

```
-- Initialize the Iceberg table in Spark SQL
CREATE TABLE default.tab_iceberg(col1 int) USING iceberg;
INSERT INTO default.tab_iceberg VALUES (1), (2), (3);
```

Step 2: Configure SynxDB Elastic for Iceberg access

On the SynxDB Elastic cluster, configure the external table:

```
-- Initializes the required extensions
CREATE EXTENSION IF NOT EXISTS datalake_fdw;
CREATE EXTENSION IF NOT EXISTS hive_connector;

-- Configures the Iceberg foreign data wrapper
CREATE FOREIGN DATA WRAPPER hdfs_fdw_iceberg
    HANDLER datalake_fdw_handler
    VALIDATOR datalake_fdw_validator
    OPTIONS (mpp_execute 'all segments');

-- Creates the foreign server
SELECT public.create_foreign_server('iceberg_server_t', 'gpadmin', 'hdfs_fdw_iceberg',
'hdfs-cluster-1');

-- Defines the external table
CREATE FOREIGN TABLE ext_t_hdfs_iceberg(
    col1 int
)
server iceberg_server_t
OPTIONS (
    filePath '/user/hive/warehouse',
    catalog_type 'hadoop',
    server_name 'hdfs-cluster-1',
    hdfs_cluster_name 'hdfs-cluster-1',
    table_identifier 'default.tab_iceberg',
    format 'iceberg'
);
```

Step 3: Read data

On the SynxDB Elastic cluster, query the Iceberg data:

```
-- Executes a data retrieval query
SELECT * FROM ext_t_hdfs_iceberg;
 col1
------
    1
    2
    3
(3 rows)
```

3.5 Load Data from Hive Data Warehouse

Hive data warehouse is built on the HDFS of the Hadoop cluster, so the data in the Hive data warehouse is also stored in HDFS. Currently, SynxDB Elastic supports writing data to and reading data from HDFS (see *Load Data from Object Storage and HDFS*) as well as reading data from Hive via the Hive Connector.

The Hive Connector loads tables from the Hive cluster as foreign tables in SynxDB Elastic, which store the paths to the data in HDFS. `datalake_fdw` reads data from these external tables, thus loading data from Hive into SynxDB Elastic.

The general steps to use the Hive Connector are as follows.

3.5.1 Step 1. Configure Hive and HDFS information on SynxDB Elastic

On containerized SynxDB Elastic, you need to create configuration files. The general steps are as follows:

1. Add the Hive MetaStore Service domain name and the namenode domain name to the CoreDNS server. If the two services use IP addresses, skip this step.
2. Update `gphive.conf` and `gphdfs.conf` in `connector-config` to add the Hive and HDFS information to your account namespace.
3. If the Hive cluster uses Kerberos for authentication, configures `krb5.conf` and `keytab` to add information required for Kerberos authentication.

Example

1. If the Hive MetaStore Service and namenode do not use IP address, add the Hive MetaStore Service domain name and the namenode domain name to the `coredns` server in the target namespace (for example `kube-system`). You might need to rebuild the `coredns` service to apply the changes.

```
kubectl -nkube-system edit cm coredns
```

```
apiVersion: v1
kind: ConfigMap
metadata:
name: coredns
```

(continues on next page)

(continued from previous page)

```
data:
Corefile: |
   .:5353 {
      hosts {
         10.13.9.156 10-13-9-156
         fallthrough
      }
   }
```

2. Configure `gphive.conf` and `gphdfs.conf` in `connector-config` in your account namespace.

```
kubectl edit configmap connector-config -n <your-account-namespace>
```

The following are examples of `gphive.conf` and `gphdfs.conf` in different authentication modes.

> **Note**
>
> - You need to replace the configuration options with your own ones. For the detailed description of each option, see *Configuration options*.
> - In the configuration files, configuration options under cluster names must be indented with 4 spaces to align with the cluster name lines. For example, in the following example, the configuration options (such as `hdfs_namenode_host` and `hdfs_namenode_port`) under `hive-cluster-1` must be indented with 4 spaces.

- For `simple` authentication mode with a single cluster.

```
apiVersion: v1
kind: ConfigMap
metadata:
name: connector-config
data:
gphdfs.conf: |
   hdfs-cluster-1:
       # namenode host
       hdfs_namenode_host: 10-13-9-156
```

(continues on next page)

(continued from previous page)

```
      # name port
      hdfs_namenode_port: 9000
      # authentication method
      hdfs_auth_method: simple
      # rpc protection
      hadoop_rpc_protection: authentication
gphive.conf: |
   hive-cluster-1:
      uris: thrift://10-13-9-156:9083
      auth_method: simple
```

- For `kerberos` authentication mode with 2 clusters for high availability.

```
apiVersion: v1
kind: ConfigMap
metadata:
name: connector-config
data:
gphdfs.conf: |
   hdfs-cluster-1:
      hdfs_namenode_host: mycluster
      hdfs_namenode_port: 9000
      hdfs_auth_method: kerberos
      krb_principal: hdfs/10-13-9-156@EXAMPLE.COM
      krb_principal_keytab: /etc/kerberos/keytab/hdfs.keytab
      is_ha_supported: true
      hadoop_rpc_protection: authentication
      data_transfer_protocol: true
      dfs.nameservices: mycluster
      dfs.ha.namenodes.mycluster: nn1,nn2
      dfs.namenode.rpc-address.mycluster.nn1: 10.13.9.156:9000
      dfs.namenode.rpc-address.mycluster.nn2: 10.13.9.157:9000
      dfs.client.failover.proxy.provider.mycluster: org.apache.hadoop.hdfs.
server.namenode.ha.ConfiguredFailoverProxyProvider
gphive.conf: |
   hive-cluster-1:
      uris: thrift://10.13.9.156:9083,thrift://10.13.9.157:9083
      auth_method: kerberos
      krb_service_principal: hive/10-13-9-156@EXAMPLE.COM
      krb_client_principal: hive/10-13-9-156@EXAMPLE.COM
```

(continues on next page)

(continued from previous page)

```
      krb_client_keytab: /etc/kerberos/keytab/hive.keytab
```

3. If the target Hive cluster uses Kerberos for authentication, in addition to `gphive.conf` and `gphdfs.conf`, you also need to configure `kerberos-config` and `keytab` that exist in the proxy and all segments.

   - To configure `kerberos-config`, run `kubectl -n<account namesapce> edit cm kerberos-config`. The following is an example of `kerberos-config` to be configured. You can get the configuration information from the `krb5.conf` file of the target Hive cluster.

```
apiVersion: v1
kind: ConfigMap
data:
   krb5.conf: |
      [logging]
       default = FILE:/var/log/krb5libs.log
       kdc = FILE:/var/log/krb5kdc.log
       admin_server = FILE:/var/log/kadmind.log
      [libdefaults]
       dns_lookup_realm = false
       ticket_lifetime = 24h
       renew_lifetime = 7d
       forwardable = true
       rdns = false
       pkinit_anchors = FILE:/etc/pki/tls/certs/ca-bundle.crt
       default_realm = EXAMPLE.COM
      [realms]
       EXAMPLE.COM = {
       kdc = 10.13.9.156
       admin_server = 10.13.9.156
      }
      [domain_realm]
       .example.com = EXAMPLE.COM
       example.com = EXAMPLE.COM
```

   - To configure `keytab`, you first need to get the `hdfs.keytab` and `hive.keytab` files, and run commands in the same directory to load the files into the cluster. For example:

```
# Loads hdfs.keytab into the cluster.
kubectl -n<account_namespace> get secret kerberos-keytab -o json | jq --arg
new_value "$(base64 -i hdfs.keytab)" '.data["hdfs.keytab"] = $new_value' |
kubectl -n<account_namespace> apply -f -

# Loads hive.keytab into the cluster.
kubectl -n<account_namespace> get secret kerberos-keytab -o json | jq --arg
new_value "$(base64 -i hive.keytab)" '.data["hive.keytab"] = $new_value' |
kubectl -n<account_namespace> apply -f -
```

After the loading, use the following commands to check the validity of `hdfs.keytab` and `hive.keytab`. The `keytab` is stored in `/etc/kerberos/keytab/`.

```
kinit -k -t hdfs.keytab hdfs/10-13-9-156@EXAMPLE.COM
kinit -k -t hive.keytab hive/10-13-9-156@EXAMPLE.COM
```

Configuration options

This section introduces the detailed description of the configuration options of `gphive.conf` and `gphdfs.conf`.

> **Attention**
>
> The default port for the `data_lake` agent has been changed from `5888` to `3888` to avoid conflict with PXF.

gphive.conf

You can get these configuration information from the `hive-site.xml` file of the target Hive cluster.

Item name	Description	Default value
uris	The listening address of Hive Metastore Service (the HMS hostname).	/
auth_method	The authentication method for Hive Metastore Service: `simple` or `kerberos`.	simple
krb_service_principal	The service principal required for the Kerberos authentication of Hive Metastore Service. When using the HMS HA feature, you need to configure the instance in the principal as `_HOST`, for example, `hive/_HOST@EXAMPLE`.	
krb_client_principal	The client principal required for the Kerberos authentication of Hive Metastore Service.	
krb_client_keytab	The keytab file of the client principal required for the Kerberos authentication of Hive Metastore Service.	
debug	The debug flag of Hive Connector: true or false.	false

gphdfs.conf

You can get these configuration information from the `hive-site.xml` and `hdfs-site.xml` files of the target Hive cluster.

Option name	Description	Default Value
hdfs_namenode_host	Configures the host information of HDFS. For example, `hdfs://mycluster`, where `hdfs://` can be omitted.	/
hdfs_namenode_port	Configures the port information of HDFS. If not configured, the default port `9000` is used.	`9000`
hdfs_auth_method	Configures the HDFS authentication method. Uses `simple` for regular HDFS. Uses `kerberos` for HDFS with Kerberos.	/
krb_principal	Kerberos principal. This is set when `hdfs_auth_method` is set to “kerberos” .	/
krb_principal_keytab	The location where the user-generated keytab is placed.	/
hadoop_rpc_protection	Should match the configuration in `hdfs-site.xml` of the HDFS cluster.	/
data_transfer_protocol	When the HDFS cluster is configured with Kerberos, there are two different methods: 1. privileged resources. 2. SASL RPC data transfer protection and SSL for HTTP. If it is the second method, “SASL” , you need to set `data_transfer_protocol` to `true` here.	/
is_ha_supported	Sets whether to use `hdfs-ha`. The value of `true` means to use `hdfs-ha`, while `false` means not to use. The default value is `false`.	`false`

3.5.2 Step 2. Create foreign data wrapper and Hive Connector extension

Before synchronization, load the `datalake_fdw` extension used for reading HDFS, and create the foreign data wrapper for reading external tables.

1. Create the necessary extensions.

```
CREATE EXTENSION synxdb;
CREATE EXTENSION dfs_tablespace;
CREATE EXTENSION gp_toolkit;
CREATE EXTENSION datalake_fdw;
```

2. Create the foreign data wrapper.

```
CREATE FOREIGN DATA WRAPPER datalake_fdw
HANDLER datalake_fdw_handler
VALIDATOR datalake_fdw_validator
OPTIONS (mpp_execute 'all segments');
```

3. Before calling the function, you need to load the Hive Connector extension.

```
CREATE EXTENSION hive_connector;
```

3.5.3 Step 3. Create server and user mapping

After creating the foreign data wrapper and Hive Connector, you need to create the server and user mapping, as shown in the following example:

```
SELECT public.create_foreign_server('sync_server', 'gpadmin', 'datalake_fdw', 'hdfs-
cluster-1');
```

In the above example, the `create_foreign_server` function takes the form as follows:

```
create_foreign_server(serverName,
                      userMapName,
                      dataWrapName,
                      hdfsClusterName);
```

This function creates a server and user mapping pointing to an HDFS cluster, which can be used by the Hive Connector to create foreign tables. The `datalake_fdw` uses the server configuration to read data from the corresponding HDFS cluster when accessing external tables.

The parameters in the function are explained as follows:

- `serverName`: The name of the server to be created.
- `userMapName`: The name of the user to be created on the server.
- `dataWrapName`: The name of the data wrapper used for reading HDFS data.
- `hdfsClusterName`: The name of the HDFS cluster where the Hive cluster is located, as specified in the configuration file.

Tip

By default, the `datalake_fdw` accesses HDFS using the system role `gpadmin`. You can use the user option in `CREATE USER MAPPING` to control which HDFS user will be used when accessing the file system. This allows finer access control to HDFS resources.

Example:

```
CREATE SERVER foreign_server
FOREIGN DATA WRAPPER datalake_fdw
OPTIONS (
   protocol 'hdfs',
   hdfs_namenodes 'hadoop-nn',
   hdfs_port '9000',
   hdfs_auth_method 'simple',
   hadoop_rpc_protection 'authentication');

CREATE USER MAPPING FOR current_user SERVER foreign_server
OPTIONS (user 'hdfs_reader');
```

In this example, the HDFS storage will be accessed with the `hdfs_reader` user rather than the default `gpadmin`. This method is recommended for managing access permissions in multi-tenant or multi-user environments.

3.5.4 Step 4. Sync Hive objects to SynxDB Elastic

Syncing a Hive table

To sync a table from Hive to SynxDB Elastic, see the following example:

```
-- Syncs Hive tables in psql.

gpadmin=# select public.sync_hive_table('hive-cluster-1', 'mytestdb', 'weblogs',
'hdfs-cluster-1', 'myschema.weblogs', 'sync_server');
 sync_hive_table
-----------------
 t
(1 row)
```

The above example uses the `sync_hive_table` function to perform the synchronization. The general form of the function is as follows:

```
sync_hive_table(hiveClusterName,
                hiveDatabaseName,
                hiveTableName,
                hdfsClusterName,
                destTableName,
                serverName,
                forceSync);
```

This function syncs a table to SynxDB Elastic, with both non-forced and forced modes available. When `forceSync` is set to `true`, the sync is forced, which means that if a table with the same name already exists in SynxDB Elastic, the existing table is dropped before syncing. If the `forceSync` parameter is not provided or is set to `false`, an error will occur if a table with the same name exists.

The parameters are explained as follows:

- `hiveClusterName`: The name of the Hive cluster where the table to be synced is located, as specified in the configuration file.
- `hiveDatabaseName`: The name of the database in Hive where the table to be synced belongs.
- `hiveTableName`: The name of the table to be synced.
- `hdfsClusterName`: The name of the HDFS cluster where the Hive cluster is located, as specified in the configuration file.
- `destTableName`: The name of the table in SynxDB Elastic where the data will be synced.

- `serverName`: The name of the server to be used when creating the foreign table with the `datalake_fdw` extension.
- `forceSync`: Optional parameter. Default value is `false`. Indicates whether the sync should be forced.

Sync a partitioned Hive table using sync_hive_partition_table

SynxDB Elastic supports synchronizing only the latest partition of a Hive table using the `sync_hive_partition_table` function. This function is used to sync a single partition specified by the highest-level partition key (for example, `prov` if the table is partitioned by `prov`, `month`, and `day`). It does not support specifying lower-level partition keys directly (such as `month` or `day`), and will return an error if you attempt to do so.

Function prototype:

```
CREATE OR REPLACE FUNCTION sync_hive_partition_table(
   hiveClusterName text,
   hiveDatabaseName text,
   hiveTableName text,
   hivePartitionValue text,
   hdfsClusterName text,
   destTableName text
) RETURNS boolean
AS '$libdir/hive_connector', 'sync_hive_partition_table'
LANGUAGE C STRICT EXECUTE ON MASTER;
```

The parameter `hivePartitionValue` means the value for the highest-level partition key. It must be the first key in the partition column list.

Example Hive table:

```
CREATE TABLE hive_table (
   id int,
   name string
)
PARTITIONED BY (
   prov int,
   month int,
   day int
);
```

Example usage:

```
SELECT sync_hive_partition_table(
   'hive-cluster-1',
   'mydb',
   'hive_table',
   '06',
   'hdfs-cluster-1',
   'myschema.hive_table_06'
);
```

This call will sync only the partition data under `prov=06`. If you try to specify values like `month=06` or `day=15`, the function will return an error.

Note: This function only supports specifying the value of the first partition key. Multi-level partition value specification is currently not supported.

Resulting external table structure:

```
CREATE TABLE mpp_table (
   id int,
   name string,
   prov int,
   month int,
   day int
)
LOCATION('gphdfs://example/prov=06/ hdfs_cluster_name=paa_cluster partitonkey=month,
day partitionvalue=06')
FORMAT 'xxx';
```

More examples

Sync a Hive text table

1. Create the following text table in Hive.

```
-- Creates the Hive table in Beeline.

CREATE TABLE weblogs
(
    client_ip           STRING,
```

(continues on next page)

(continued from previous page)

```
    full_request_date   STRING,
    day                 STRING,
    month               STRING,
    month_num           INT,
    year                STRING,
    referrer            STRING,
    user_agent          STRING
) STORED AS TEXTFILE;
```

2. Sync the text table to SynxDB Elastic.

```
-- Syncs the Hive table in psql.

gpadmin=# select public.sync_hive_table('hive-cluster-1', 'mytestdb', 'weblogs',
'hdfs-cluster-1', 'myschema.weblogs', 'sync_server');
sync_hive_table
-----------------
t
(1 row)
```

3. Query the external table.

```
SELECT * FROM myschema.weblogs LIMIT 10;
```

Sync a Hive ORC table

1. Create an ORC table in Hive.

```
-- Creates the Hive table in Beeline.
CREATE TABLE test_all_type
(
    column_a tinyint,
    column_b smallint,
    column_c int,
    column_d bigint,
    column_e float,
    column_f double,
    column_g string,
    column_h timestamp,
    column_i date,
```

(continues on next page)

(continued from previous page)

```
    column_j char(20),
    column_k varchar(20),
    column_l decimal(20, 10)
) STORED AS ORC;
```

2. Sync the ORC table to SynxDB Elastic:

```
-- Syncs the Hive table in psql.

gpadmin=# select public.sync_hive_table('hive-cluster-1', 'mytestdb', 'test_all_
type', 'hdfs-cluster-1', 'myschema.test_all_type', 'sync_server');
sync_hive_table
-----------------
t
(1 row)
```

3. Query the external table.

```
SELECT * FROM myschema.test_all_type LIMIT 10;
```

Sync a Hive ORC partitioned table

1. Create an ORC partitioned table in Hive.

```
-- Creates the Hive table in Beeline.

CREATE TABLE test_partition_1_int
(
    a tinyint,
    b smallint,
    c int,
    d bigint,
    e float,
    f double,
    g string,
    h timestamp,
    i date,
    j char(20),
    k varchar(20),
    l decimal(20, 10)
```

(continues on next page)

(continued from previous page)

```
)
PARTITIONED BY
(
    m int
)
STORED AS ORC;
INSERT INTO test_partition_1_int VALUES (1, 1, 1, 1, 1, 1, '1', '2020-01-01
01:01:01', '2020-01-01', '1', '1', 10.01, 1);
INSERT INTO test_partition_1_int VALUES (2, 2, 2, 2, 2, 2, '2', '2020-02-02
02:02:02', '2020-02-01', '2', '2', 11.01, 2);
INSERT INTO test_partition_1_int VALUES (3, 3, 3, 3, 3, 3, '3', '2020-03-03
03:03:03', '2020-03-01', '3', '3', 12.01, 3);
INSERT INTO test_partition_1_int VALUES (4, 4, 4, 4, 4, 4, '4', '2020-04-04
04:04:04', '2020-04-01', '4', '4', 13.01, 4);
INSERT INTO test_partition_1_int VALUES (5, 5, 5, 5, 5, 5, '5', '2020-05-05
05:05:05', '2020-05-01', '5', '5', 14.01, 5);
```

2. Sync the ORC partitioned table to SynxDB Elastic.

```
-- psql syncs the Hive partitioned tables as one foreign table.

gpadmin=# select public.sync_hive_table('hive-cluster-1', 'mytestdb', 'test_
partition_1_int', 'hdfs-cluster-1', 'myschema.test_partition_1_int', 'sync_server
');
sync_hive_table
-----------------
t
(1 row)
```

3. Query the external table.

```
SELECT * FROM myschema.test_partition_1_int LIMIT 10;
```

3.5.5 Supported usage and limitations

Supported Hive file formats

You can load files in TEXT, CSV, ORC, or PARQUET formats from Hive into SynxDB Elastic.

Data type mapping

The following table shows the one-to-one mapping between table data types on a Hive cluster and table data types in SynxDB Elastic.

Hive	SynxDB Elastic
binary	bytea
tinyint	smallint
smallint	smallint
int	int
bigint	bigint
float	float4
double	double precision
string	text
timestamp	timestamp
date	date
char	char
varchar	varchar
decimal	decimal

Usage limitations

- Synchronizing Hive external tables is not supported.
- Synchronizing Hive table statistics is not supported.
- SynxDB Elastic can read data from HDFS and write data to HDFS, but the written data cannot be read by Hive.
- When using `sync_hive_partition_table`, only the first-level partition key is supported. Specifying a value from a secondary or lower-level partition key will result in an error.

Note

Q: How is write and update on HDFS synchronized to SynxDB Elastic? Are there any limitations?

A: The data is still stored in HDFS, and the Foreign Data Wrapper only reads the data from HDFS.

3.6 Load Data Using `gpfdist`

To load data from local host files or files accessible via internal network, you can use the `gpfdist` protocol in the `CREATE EXTERNAL TABLE` statement. gpfdist is a file server utility that runs on a host other than the SynxDB Elastic coordinator or standby coordinator. `gpfdist` serves files from a directory on the host to SynxDB Elastic segments.

When external data is served by gpfdist, all segments in the SynxDB Elastic system can read or write external table data in parallel.

The supported data formats are:

- CSV and TXT
- Any delimited text format supported by the `FORMAT` clause

The general procedure for loading data using `gpfdist` is as follows:

1. Install gpfdist on a host other than the SynxDB Elastic coordinator or standby coordinator. See *Install gpfdist*.
2. Start gpfdist on the host. See *Start and stop gpfdist*.
3. Create an external table using the `gpfdist` protocol. See *Examples for using gpfdist with external tables*.

3.6.1 About gpfdist

Before using gpfdist, you might need to know how it works. This section provides an overview of gpfdist and how to use it with external tables.

About gpfdist and external tables

The `gpfdist` file server utility is located in the `$GPHOME/bin` directory on your SynxDB Elastic coordinator host and on each segment host. When you start a `gpfdist` instance you specify a listen port and the path to a directory containing files to read or where files are to be written. For example, this command runs `gpfdist` in the background, listening on port `8801`, and serving files in the `/home/gpadmin/external_files` directory:

```
$ gpfdist -p 8801 -d /home/gpadmin/external_files &
```

The `CREATE EXTERNAL TABLE` command `LOCATION` clause connects an external table definition

to one or more `gpfdist` instances. If the external table is readable, the `gpfdist` server reads data records from files from in specified directory, packs them into a block, and sends the block in a response to a SynxDB Elastic segment's request. The segments unpack rows that they receive and distribute the rows according to the external table's distribution policy. If the external table is a writable table, segments send blocks of rows in a request to gpfdist and gpfdist writes them to the external file.

External data files can contain rows in CSV format or any delimited text format supported by the `FORMAT` clause of the `CREATE EXTERNAL TABLE` command.

For readable external tables, `gpfdist` uncompresses `gzip` (`.gz`) and `bzip2` (`.bz2`) files automatically. You can use the wildcard character (`*`) or other C-style pattern matching to denote multiple files to read. External files are assumed to be relative to the directory specified when you started the `gpfdist` instance.

About gpfdist setup and performance

You can run `gpfdist` instances on multiple hosts and you can run multiple `gpfdist` instances on each host. This allows you to deploy `gpfdist` servers strategically so that you can attain fast data load and unload rates by utilizing all of the available network bandwidth and SynxDB Elastic's parallelism.

- Allow network traffic to use all ETL host network interfaces simultaneously. Run one instance of gpfdist for each interface on the ETL host, then declare the host name of each NIC in the `LOCATION` clause of your external table definition (see *Examples for Creating External Tables*).
- Divide external table data equally among multiple gpfdist instances on the ETL host. For example, on an ETL system with two NICs, run two gpfdist instances (one on each NIC) to optimize data load performance and divide the external table data files evenly between the two gpfdist servers.

Tip

Use pipes (`|`) to separate formatted text when you submit files to gpfdist. SynxDB Elastic encloses comma-separated text strings in single or double quotes. gpfdist has to remove the quotes to parse the strings. Using pipes to separate formatted text avoids the extra step and improves performance.

Control segment parallelism

The `gp_external_max_segs` server configuration parameter controls the number of segment instances that can access a single gpfdist instance simultaneously. 64 is the default. You can set the number of segments such that some segments process external data files and some perform other database processing. Set this parameter in the `postgresql.conf` file of your coordinator instance.

3.6.2 Step 1. Install gpfdist

gpfdist is installed in `$GPHOME/bin` of your SynxDB Elastic coordinator host installation. Run gpfdist on a machine other than the SynxDB Elastic coordinator or standby coordinator, such as on a machine devoted to ETL processing. Running gpfdist on the coordinator or standby coordinator can have a performance impact on query execution.

3.6.3 Step 2. Start and stop gpfdist

You can start gpfdist in your current directory location or in any directory that you specify. The default port is `8080`.

From your current directory, type:

```
gpfdist &
```

From a different directory, specify the directory from which to serve files, and optionally, the HTTP port to run on.

To start gpfdist in the background and log output messages and errors to a log file:

```
$ gpfdist -d /var/load_files -p 8081 -l /home/`gpadmin`/log &
```

For multiple gpfdist instances on the same ETL host, use a different base directory and port for each instance. For example:

```
$ gpfdist -d /var/load_files1 -p 8081 -l /home/`gpadmin`/log1 &
$ gpfdist -d /var/load_files2 -p 8082 -l /home/`gpadmin`/log2 &
```

The logs are saved in `/home/gpadmin/log`.

3.6.4 Step 3. Use gpfdist with external tables to load data

The following examples show how to use gpfdist when creating an external table to load data into SynxDB Elastic.

> **Tip**
>
> When using IPv6, always enclose the numeric IP addresses in square brackets.

Example 1 - Run single gpfdist instance on a single-NIC machine

Creates a readable external table, ext_expenses, using the gpfdist protocol. The files are formatted with a pipe (|) as the column delimiter.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
    date date, amount float4, category text, desc1 text )
    LOCATION ('gpfdist://etlhost-1:8081/*')
FORMAT 'TEXT' (DELIMITER '|');
```

Example 2 —Run multiple gpfdist instances

Creates a readable external table, `ext_expenses`, using the gpfdist protocol from all files with the txt extension. The column delimiter is a pipe (|) and NULL (' ') is a space.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('gpfdist://etlhost-1:8081/*.txt',
             'gpfdist://etlhost-2:8081/*.txt')
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ') ;
```

Example 3 —Single gpfdist instance with error logging

Uses the gpfdist protocol to create a readable external table, `ext_expenses`, from all files with the txt extension. The column delimiter is a pipe (|) and NULL (' ') is a space.

Access to the external table is single row error isolation mode. Input data formatting errors are captured internally in SynxDB Elastic with a description of the error. You can view the errors, fix the issues, and then reload the rejected data. If the error count on a segment is greater than `5` (the `SEGMENT REJECT LIMIT` value), the entire external table operation fails and no rows are processed.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date, amount float4, category text, desc1 text )
   LOCATION ('gpfdist://etlhost-1:8081/*.txt',
             'gpfdist://etlhost-2:8082/*.txt')
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ')
   LOG ERRORS SEGMENT REJECT LIMIT 5;
```

To create the readable `ext_expenses` table from CSV-formatted text files:

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('gpfdist://etlhost-1:8081/*.txt',
             'gpfdist://etlhost-2:8082/*.txt')
   FORMAT 'CSV' ( DELIMITER ',' )
   LOG ERRORS SEGMENT REJECT LIMIT 5;
```

Example 4 - Create a writable external table with gpfdist

Creates a writable external table, · `sales_out`, that uses gpfdist to write output data to the file `sales.out`. The column delimiter is a pipe (`|`) and NULL (`' '`) is a space. The file will be created in the directory specified when you started the gpfdist file server.

```
=# CREATE WRITABLE EXTERNAL TABLE sales_out (LIKE sales)
   LOCATION ('gpfdist://etl1:8081/sales.out')
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ');
```

Chapter 4

Operate with Data

4.1 Operate with Database Objects

4.1.1 Basic Queries

This document introduce the basic queries of SynxDB Elastic.

SynxDB Elastic is a high-performance, highly parallel data warehouse developed based on PostgreSQL and Greenplum. Here are some examples of the basic query syntax.

- `SELECT`: Used to retrieve data from databases & tables.

```
SELECT * FROM employees;  -- Queries all data in the employees table.
```

- Conditional query (`WHERE`): Used to filter result sets based on certain conditions.

```
SELECT * FROM employees WHERE salary > 50000;  -- Queries employee information
with salary exceeding 50,000.
```

- `ORDER BY`: Used to sort query results by one or more columns.

```
SELECT * FROM employees ORDER BY salary DESC;  -- Sorts employee information in
descending order by salary.
```

- Aggregation functions: such as `COUNT`, `SUM`, `AVG`, `MAX`, `MIN`, used for calculating statistics

from datasets.

```
SELECT AVG(salary) FROM employees;  -- Calculates the average salary of employees.
```

- `GROUP BY`: Used in conjunction with aggregation functions to group result sets.

```
SELECT department, COUNT(*) FROM employees GROUP BY department;  -- Counts the
number of employees by department.
```

- Limit the number of results (`LIMIT`): used to limit the number of rows returned by the query result.

```
SELECT * FROM employees LIMIT 10;  -- Only queries the information of the first 10
employees.
```

- Join query (`JOIN`): used to combine data from two or more tables based on related columns.

```
SELECT employees.name, departments.name
FROM employees
JOIN departments ON employees.department_id = departments.id;  -- Queries
employees and their corresponding department names.
```

- Subquery: Nested queries in another SQL query.

```
SELECT name FROM employees
WHERE department_id IN (SELECT id FROM departments WHERE location = 'New York');
-- Queries all employees working in New York.
```

The above is just a brief overview of the basic query syntax in SynxDB Elastic. SynxDB Elastic also provides more advanced queries and functions to help developers perform complex data operations and analyses.

4.1.2 Create and Manage Warehouses

In SynxDB Elastic, warehouses are compute engines within an account to process queries. Each warehouse is composed of multiple segments, which can be scaled horizontally as needed. All warehouses within a single account have access to a shared pool of data and perform computing tasks independently. Key features of warehouses include:

- Resource management: Allows pausing, resuming, and resizing on demand.

- Scaling: Supports changing segment counts for online horizontal scaling.

This document introduces how to create and manage warehouses in SynxDB Elastic.

Check existing warehouses

To check the existing warehouses, you can use either of the methods:

- Use `psql` to run the following SQL command:

```
TABLE gp_warehouse;
```

- Use the SynxDB Elastic console. Click **Warehouses** in the left navigation menu to enter the warehouse management page, and you will see a list of existing warehouses.

Create a warehouse

To create a warehouse, you can use either of the methods:

- Use `psql` to run the following SQL command:

```
CREATE WAREHOUSE <warehouse_name> WAREHOUSE_SIZE <segment_count>;
```

  In the command above, replace `<warehouse_name>` with the name of the warehouse you want to create and `<segment_count>` with the number of segments you want to allocate to the warehouse.

  For example:

```
CREATE WAREHOUSE my_warehouse WAREHOUSE_SIZE 2;
```

- Use the SynxDB Elastic console. For details, see *Create a warehouse via console*.

Set and use a warehouse

Before updating data in a table or deleting database objects, you need to set and use a warehouse. To set a warehouse, you can use either of the methods:

- Use `psql` to run the following SQL command:

```
SET WAREHOUSE to <warehouse_name>;
```

In the command above, replace `<warehouse_name>` with the name of the warehouse you want to use.

- Use the SynxDB Elastic console. For details, see *Set and use a warehouse via console*.

Change the size of a warehouse

You can elastically scale warehouses to meet dynamic business workloads. You can use either of the methods:

- Use `psql` to run the following SQL command. Increase the value for scaling out or decrease it for scaling in based on your requirements.

```
ALTER WAREHOUSE <warehouse_name> SET WAREHOUSE_SIZE <new_segment_count>;
```

For example:

```
ALTER WAREHOUSE my_warehouse SET WAREHOUSE_SIZE 4;
```

- Use the SynxDB Elastic console. For details, see *Change the size of a warehouse via console*.

Optimize query execution for warehouses

In high-concurrency scenarios, if a data warehouse has too many segment nodes (processes), frequent process scheduling and context switching can create significant overhead, affecting query performance. In some cases, reducing the number of segments involved in a query can actually lead to better execution performance. To address such scenarios, SynxDB Elastic allows you to set the number of segments to be used for a query at the session level.

You can control the number of segments used in the current session by setting the `cloud.session_segments` parameter. The system will randomly select the specified number of segments from the current data warehouse to execute the query.

```
SET cloud.session_segments = <number_of_segments>;
```

Setting `number_of_segments` to `0` (the default) means the query will use all segments of the current data warehouse. Setting it to a positive integer `N` means the query will be executed on `N` randomly selected segments from the current data warehouse.

Note

- The value of `number_of_segments` cannot exceed the total number of segments in the current data warehouse.
- This parameter cannot be set within a transaction block (`BEGIN...COMMIT`).
- To restore the default behavior, execute `RESET cloud.session_segments;` or `SET cloud.session_segments = 0;`.

Example:

Suppose `my_warehouse` has a total of 4 segments. Now, you want to use only 2 of them to execute a query.

1. Switch to the target warehouse:

```
SET warehouse = my_warehouse;
```

2. Set the number of segments for the session:

```
SET cloud.session_segments = 2;
```

3. Execute the query. At this point, the query will run on only 2 randomly selected segments from `my_warehouse`:

```
SELECT * FROM my_table;
```

   You can use the `EXPLAIN` command to view the query plan and confirm the number of segments used for execution.

4. Restore the use of all segments:

```
RESET cloud.session_segments;
```

Stop a warehouse

To stop a warehouse, you can use either of the methods:

- Use `psql` to run the following SQL command:

```
ALTER WAREHOUSE <warehouse_name> SUSPEND;
```

- Use the SynxDB Elastic console. Click **Warehouses** in the left navigation menu to enter the warehouse management page, and then click the stop button in the **Start/Stop** column of the warehouse you want to stop.

Resume a warehouse

To resume a warehouse, you can use either of the methods:

- Use `psql` to run the following SQL command:

```
ALTER WAREHOUSE <warehouse_name> RESUME;
```

- Use the SynxDB Elastic console. Click **Warehouses** in the left navigation menu to enter the warehouse management page, and then click the start button in the **Start/Stop** column of the warehouse you want to resume.

Delete a warehouse

To delete a warehouse, you can use either of the methods:

- Use `psql` to run the following SQL command:

```
DROP WAREHOUSE <warehouse_name>;
```

- Use the SynxDB Elastic console. Click **Warehouses** in the left navigation menu to enter the warehouse management page, and then click **Delete** in the **Operation** column of the warehouse you want to delete.

4.1.3 Create and Manage Tables

SynxDB Elastic tables are similar to tables in any relational database.

Create a table

The `CREATE TABLE` command creates a table and defines its structure. When you create a table, you define:

- The columns of the table and their associated data types. See *Choose column data types*.
- Any table or column constraints to limit the data that a column or table can contain. See *Setting table and column constraints*.
- The way the table is stored on disk.
- The table partitioning strategy for large tables.

Choose column data types

The data type of a column determines the types of data values the column can contain. Choose the data type that uses the least possible space but can still accommodate your data and that best constrains the data. For example, use character data types for strings, date or timestamp data types for dates, and numeric data types for numbers.

For table columns that contain textual data, specify the data type `VARCHAR` or `TEXT`. Specifying the data type `CHAR` is not recommended. In SynxDB Elastic, the data types `VARCHAR` or `TEXT` handle padding added to the data (space characters added after the last non-space character) as significant characters, the data type `CHAR` does not.

Use the smallest numeric data type that will accommodate your numeric data and allow for future expansion. For example, using `BIGINT` for data that fits in `INT` or `SMALLINT` wastes storage space. If you expect that your data values will expand over time, consider that changing from a smaller datatype to a larger datatype after loading large amounts of data is costly. For example, if your current data values fit in a `SMALLINT` but it is likely that the values will expand, `INT` is the better long-term choice.

Use the same data types for columns that you plan to use in cross-table joins. When the data types are different, the database must convert one of them so that the data values can be compared correctly, which adds unnecessary overhead.

Set table and column constraints

You can define constraints on columns and tables to restrict the data in your tables. SynxDB Elastic support for constraints is the same as PostgreSQL with some limitations, including:

- `CHECK` constraints can refer only to the table on which they are defined.
- `FOREIGN KEY` constraints are allowed, but not enforced.
- Constraints that you define on partitioned tables apply to the partitioned table as a whole. You cannot define constraints on the individual parts of the table.

Check constraints

Check constraints allow you to specify that the value in a certain column must satisfy a Boolean (truth-value) expression. For example, to require positive product prices:

```
CREATE TABLE products
       ( product_no integer,
         name text,
         price numeric CHECK (price > 0) );
```

Not-null constraints

Not-null constraints specify that a column must not assume the null value. A not-null constraint is always written as a column constraint. For example:

```
CREATE TABLE products
 ( product_no integer NOT NULL,
   name text NOT NULL,
   price numeric );
```

Foreign keys

Foreign keys are not supported. You can declare them, but referential integrity is not enforced.

Foreign key constraints specify that the values in a column or a group of columns must match the values appearing in some row of another table to maintain referential integrity between two related tables.

Create temporary tables

A temporary table is a table that exists only during the current database session. When the session ends, the temporary table and all its data are automatically deleted. Temporary tables are very useful for storing intermediate result sets during a transaction or session. You can use the `CREATE TEMPORARY TABLE` or `CREATE TEMP TABLE` statement to create a temporary table.

Optimize temporary tables using an in-memory catalog

When a temporary table is created in SynxDB Elastic, by default, corresponding metadata records are inserted into system catalog tables (such as `pg_class` and `pg_attribute`). Although the temporary table becomes invalid after the session ends, these metadata records remain in the system tables (invisible to users), which can lead to system table bloat in scenarios where many temporary tables are created.

To solve this problem, SynxDB Elastic introduces the `inmemory_catalog` storage option. When this option is set to `true` while a temporary table is created, the table's metadata is stored in memory instead of being written to the on-disk system tables. When the session ends, this in-memory metadata is automatically destroyed along with the temporary table, thus preventing system table bloat.

Example:

```
CREATE TEMP TABLE virtualcat_test1(x int, s text) WITH (inmemory_catalog=true);

INSERT INTO virtualcat_test1 VALUES (1, 'aaa'), (2, 'bbb'), (3, 'ccc'), (4, 'ddd');

UPDATE virtualcat_test1 SET s = 'eee' WHERE x = 4;

UPDATE virtualcat_test1 SET x = 5 WHERE s = 'bbb';

DELETE FROM virtualcat_test1 WHERE x = 3;

SELECT * FROM virtualcat_test1 ORDER BY x;

DROP TABLE virtualcat_test1;
```

Note

Once the `inmemory_catalog` option is specified when creating a table, it cannot be changed later using the `ALTER TABLE` statement.

4.1.4 Create and Manage Views

In SynxDB Elastic, views enable you to save frequently used or complex queries, then access them in a `SELECT` statement as if they were a table. A view is not physically materialized on disk: the query runs as a subquery when you access the view.

Create views

The `CREATE VIEW` command defines a view of a query. For example:

```
CREATE VIEW comedies AS SELECT * FROM films WHERE kind = 'comedy';
```

Drop views

The `DROP VIEW` command removes a view. For example:

```
DROP VIEW topten;
```

The `DROP VIEW ... CASCADE` command also removes all dependent objects. As an example, if another view depends on the view which is about to be dropped, the other view will be dropped as well. Without the `CASCADE` option, the `DROP VIEW` command will fail.

Best practices when creating views

When defining and using a view, remember that a view is just a SQL statement and is replaced by its definition when the query is run.

These are some common uses of views:

- They allow you to have a recurring SQL query or expression in one place for easy reuse.
- They can be used as an interface to abstract from the actual table definitions, so that you can reorganize the tables without having to modify the interface.
- If a subquery is associated with a single query, consider using the `WITH` clause of the `SELECT` command instead of creating a seldom-used view.

In general, these uses do not require nesting views, which means defining views based on other

views.

These are two patterns of creating views that tend to be problematic because the view's SQL is used during query execution:

- Defining many layers of views so that your final queries look deceptively simple.

  Problems arise when you try to enhance or troubleshoot queries that use the views, for example, by examining the execution plan. The query's execution plan tends to be complicated and it is difficult to understand and improve.

- Defining a denormalized "world" view.

  A view that joins a large number of database tables and is used for a wide variety of queries. Performance issues can occur for some queries that use the view for certain `WHERE` conditions, while other `WHERE` conditions work well.

4.1.5 Create and Manage Materialized Views

In SynxDB Elastic, materialized views are similar to views. A materialized view enables you to save a frequently used or complex query and then access the query results in a `SELECT` statement as if they were a table. Materialized views persist the query results in a table-like form.

Although accessing the data stored in a materialized view can be much faster than accessing the underlying tables directly or through a regular view, the data is not always current. The materialized view data cannot be directly updated. To refresh the materialized view data, use the `REFRESH MATERIALIZED VIEW` command.

The query used to create the materialized view is stored in exactly the same way that a view's query is stored. For example, you can create a materialized view that quickly displays a summary of historical sales data for situations where having incomplete data for the current date is acceptable.

```
CREATE MATERIALIZED VIEW sales_summary AS
  SELECT seller_no, invoice_date, sum(invoice_amt)::numeric(13,2) as sales_amt
    FROM invoice
    WHERE invoice_date < CURRENT_DATE
    GROUP BY seller_no, invoice_date;
```

The materialized view might be useful for displaying a graph in the dashboard created for salespeople. You can schedule a job to update the summary information each night using the

following command:

```
REFRESH MATERIALIZED VIEW sales_summary;
```

The information about a materialized view in the SynxDB Elastic system catalogs is exactly the same as it is for a table or view. A materialized view is a relation, just like a table or a view. When a materialized view is referenced in a query, the data is returned directly from the materialized view, just like from a table. The query in the materialized view definition is only used for populating the materialized view.

If you can tolerate periodically updating the materialized view data, you can get great performance benefits from the view.

One use of a materialized view is to allow faster access to data brought in from an external data source, such as an external table or a foreign data wrapper.

If a subquery is associated with a single query, consider using the `WITH` clause of the `SELECT` command instead of creating a seldom-used materialized view.

Create materialized views

The `CREATE MATERIALIZED VIEW` command defines a materialized view based on a query.

```
CREATE MATERIALIZED VIEW us_users AS
SELECT u.id, u.name, a.zone
FROM users u, address a
WHERE a.country = 'USA';
```

> **Note**
>
> When a materialized view is created with an `ORDER BY` or `SORT` clause, this sorting is applied only at the time of the view's initial creation. Subsequent refreshes of the materialized view do not maintain this order, because the view is essentially a static snapshot of data and does not dynamically update or preserve the sorting with new data insertions.

Create asynchronous materialized views

To avoid the performance overhead of maintaining incremental materialized views synchronously during high-frequency data writes, SynxDB Elastic allows you to create asynchronous incremental materialized views. The refresh tasks for these views run asynchronously in the background.

You can use the `CREATE INCREMENTAL MATERIALIZED VIEW` statement with the `REFRESH DEFERRED` clause to create an asynchronous incremental materialized view. The syntax is as follows:

```
CREATE INCREMENTAL MATERIALIZED VIEW view_name
[ REFRESH DEFERRED ]
[ SCHEDULE 'interval' ]
AS query;
```

- `REFRESH DEFERRED`: A mandatory clause that enables the asynchronous refresh mode.
- `SCHEDULE 'interval'`: An optional clause to specify the refresh interval for the background task, for example, `'10 seconds'`. The interval can be set from 1 to 59 seconds. If not specified, it defaults to refreshing after 30 seconds.

For example, to create an asynchronous incremental materialized view that refreshes automatically every 10 seconds:

```
CREATE INCREMENTAL MATERIALIZED VIEW mv_ivm_agg
REFRESH DEFERRED SCHEDULE '10 seconds'
AS
SELECT i, SUM(j), COUNT(*)
FROM mv_base_a
WHERE j > 10
GROUP BY i;
```

For materialized views that contain aggregate functions (such as `SUM`, `COUNT`, `AVG`, `MIN`, `MAX`), you can also choose to store intermediate aggregate results to optimize refresh performance. This can be achieved using the `WITH (partial_agg = true)` parameter.

```
CREATE INCREMENTAL MATERIALIZED VIEW mv_ivm_agg WITH (partial_agg=true)
REFRESH DEFERRED SCHEDULE '10 seconds'
AS ...
```

Create materialized views on partitioned tables

SynxDB Elastic supports creating materialized views on partitioned tables. When the base table is a partitioned table, the syntax for creating a materialized view is identical to that for a regular table. The database automatically handles interactions with the underlying partitions. This allows you to combine the management benefits of partitioned tables with the query performance advantages of materialized views.

Here is an example of creating an incremental materialized view on a two-level partitioned table.

1. Create a partitioned table. Create a two-level partitioned table `two_level_pt_ivm` that is range-partitioned by columns `b` and `c`.

```
CREATE TABLE two_level_pt_ivm (
    a int,
    b int,
    c int
)
PARTITION BY RANGE (b)
    SUBPARTITION BY RANGE (c)
    SUBPARTITION TEMPLATE (
        START (11) END (12) EVERY (1)
    )
    (
        START (1) END (2) EVERY (1)
    );
```

2. Create a materialized view on the partitioned table. Create an asynchronous incremental materialized view `mv_ivm35` based on the partitioned table `two_level_pt_ivm`.

```
CREATE INCREMENTAL MATERIALIZED VIEW mv_ivm35
WITH (partial_agg=true) REFRESH DEFERRED
AS SELECT b, sum(a) FROM two_level_pt_ivm GROUP BY b;
```

3. Insert data and refresh. Insert data into the base table, then manually refresh the materialized view to update the aggregate results.

```
INSERT INTO two_level_pt_ivm (a, b, c) VALUES (1, 1, 11);
REFRESH MATERIALIZED VIEW mv_ivm35;
```

4. Query the materialized view for fast aggregate results.

```
SELECT * FROM mv_ivm35 ORDER BY 1;
```

5. When dropping the base table, use the `CASCADE` option to automatically drop the dependent materialized view `mv_ivm35`.

```
DROP TABLE two_level_pt_ivm CASCADE;
```

Refresh or deactivate materialized views

The `REFRESH MATERIALIZED VIEW` command updates the materialized view data.

```
REFRESH MATERIALIZED VIEW us_users;
```

With the `WITH NO DATA` clause, the current data is removed, no new data is generated, and the materialized view is left in an unscannable state. An error is returned if a query attempts to access an unscannable materialized view.

```
REFRESH MATERIALIZED VIEW us_users WITH NO DATA;
```

Refresh incremental materialized views

For incremental materialized views, in addition to the standard `REFRESH MATERIALIZED VIEW` (which performs a full refresh), more fine-grained refresh operations are supported.

- Background auto-refresh: For asynchronous materialized views, a background process automatically applies new data using the `REFRESH INCREMENTAL MATERIALIZED VIEW CONCURRENTLY` command. The `CONCURRENTLY` option ensures that the refresh operation does not block read queries on the materialized view.
- Manual merge: You can manually execute the following command to merge the intermediate results of an incremental materialized view.

```
COMBINE INCREMENTAL MATERIALIZED VIEW view_name;
```

- Automatic full refresh trigger: It is important to note that when `DELETE`, `UPDATE`, or `TRUNCATE` operations occur on the base table of an incremental materialized view, the system automatically triggers a full refresh (equivalent to executing `REFRESH MATERIALIZED VIEW`) to ensure data consistency.

Drop materialized views

The `DROP MATERIALIZED VIEW` command removes a materialized view definition and data. For example:

```
DROP MATERIALIZED VIEW us_users;
```

The `DROP MATERIALIZED VIEW ... CASCADE` command also removes all dependent objects. For example, if another materialized view depends on the materialized view which is about to be dropped, the other materialized view will be dropped as well. Without the `CASCADE` option, the `DROP MATERIALIZED VIEW` command fails.

Automatically use materialized views for query optimization

SynxDB Elastic supports automatically using materialized views during the query planning phase to compute part or all of a query (also known as AQUMV). For more details, see *Automatically Use Materialized Views for Query Optimization*.

4.1.6 Insert, Update, and Delete Row Data

This document introduces how to manipulate row data in SynxDB Elastic, including:

- *Inserting rows*
- *Updating existing rows*
- *Deleting rows*
- *Truncating a table*
- *Vacuuming the database*

Insert rows

Use the `INSERT` command to create rows in a table. This command requires the table name and a value for each column in the table; you might optionally specify the column names in any order. If you do not specify column names, list the data values in the order of the columns in the table, separated by commas.

For example, to specify the column names and the values to insert:

```
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

To specify only the values to insert:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

Usually, the data values are literals (constants), but you can also use scalar expressions. For example:

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod <
'2016-05-07';
```

You can insert multiple rows in a single command. For example:

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

Insert rows into partitioned tables

To insert data into a partitioned table, you are expected to specify the root partitioned table that was created with the `CREATE TABLE` command. Directly specifying a leaf partition in an `INSERT` command is not supported, and attempting to do so will cause an error, because leaf partitions are invisible to users and data insertion is managed automatically by the database system.

An error will be returned if the data being inserted does not fit the range of any existing partitions (for example, the specified key value does not match any partition rules).

To ensure data is correctly inserted into a partitioned table, you only need to specify the root partitioned table in your `INSERT` statement. The database system will automatically insert the data row into the appropriate leaf partition based on the partition key. If a data row does not conform to the range of any leaf partition, the database will return an error.

Example:

```
-- Inserting data into the root partitioned table
INSERT INTO sales (sale_id, product_no, year, amount) VALUES (1, 'Cheese', 2021, 9.
99);
```

The above statement will automatically insert the data row into the correct partition based on the value of the year column. You should not, and need not, attempt to directly specify any leaf partition for data insertion.

Insert rows into append-optimized tables

To insert large amounts of data, use external tables or the `COPY` command. These load mechanisms are more efficient than `INSERT` for inserting many rows.

The storage model of append-optimized tables in SynxDB Elastic is designed for efficient bulk data loading rather than single row `INSERT` statements. For high-volume data insertions, it is recommended to use batch loading methods such as the `COPY` command. SynxDB Elastic can support multiple concurrent `INSERT` transactions on append-optimized tables; however, this capability is typically intended for batch insertions rather than single-row operations.

Update existing rows

The `UPDATE` command updates rows in a table. You can update all rows, a subset of all rows, or individual rows in a table. You can update each column separately without affecting other columns.

To perform an update, you need:

- The name of the table and columns to update.
- The new values of the columns.
- One or more conditions specifying the row or rows to be updated.

For example, the following command updates all products that have a price of *5* to have a price of *10*:

```
UPDATE products SET price = 10 WHERE price = 5;
```

Delete rows

The `DELETE` command deletes rows from a table. Specify a `WHERE` clause to delete rows that match certain criteria. If you do not specify a `WHERE` clause, all rows in the table are deleted. The result is a valid, but empty, table. For example, to remove all rows from the products table that have a price of *10*:

```
DELETE FROM products WHERE price = 10;
```

To delete all rows from a table:

```
DELETE FROM products;
```

Truncate a table

Use the `TRUNCATE` command to quickly remove all rows in a table. For example:

```
TRUNCATE mytable;
```

This command empties a table of all rows in one operation. Note that in SynxDB Elastic, the `TRUNCATE` command will affect inherited child tables by default, even without using the `CASCADE` option. In addition, because SynxDB Elastic does not support foreign key constraints, the `TRUNCATE` command will not trigger any `ON DELETE` actions or rewrite rules. The command truncates only rows in the named table.

Vacuum the database

Deleted or updated data rows occupy physical space on disk even though new transactions cannot see them. Periodically running the `VACUUM` command removes these expired rows. For example:

```
VACUUM mytable;
```

The `VACUUM` command collects table-level statistics such as the number of rows and pages. Vacuum all tables after loading data, including append-optimized tables.

You need to use the `VACUUM`, `VACUUM FULL`, and `VACUUM ANALYZE` commands to maintain the data in a SynxDB Elastic especially if updates and deletes are frequently performed on your database data.

4.1.7 Use Tags to Manage Database Objects

A tag is a database-level object that helps users label, classify, and manage other objects in the database. using tags, you can add custom labels to database objects such as databases, users, and tables, and warehouses,making it easier to manage and monitor different objects in the database.

You can use tags in the database mainly used for the following purposes:

- Compliance monitoring for sensitive data: Tags can mark sensitive data to help administrators track and protect data, thus ensuring compliance.
- Data governance and resource usage management: Tags can classify and label different types of database objects, which helps discover data, protect it, and monitor resource usage.

Tags are stored as key-value pairs, with a tag name and its value making up the pair. You can assign the same tag to multiple database objects with different values for flexible management.

SynxDB Elastic supports and enables tags by default, with no extra setup needed.

Features of tags

- You can assign tags to any object in the database, such as databases, users, tables, and warehouses.
- You can assign tags when creating a database object with the `CREATE <object>` statement, or add or change tags later using the `ALTER <object>` statement.
- You can assign the same tag to different types of objects (like databases and tables) at the same time, and the tag value can be the same or different each time.

Usage scenarios

The tag is suitable for these situations:

- Object classification: Tags can be used to classify database objects. For example, you can use tags to tell the difference between objects in a development environment and a production environment. This makes it easier to quickly find and manage database objects in different environments.
- Permission management: By giving objects-specific tags, you can mark objects with special permissions or sensitive data. This helps administrators be more accurate when checking data, giving permissions, and managing compliance.
- Version control and status tracking: Tags can also be used to mark the version or status of database objects. This helps teams track the history of changes or see the database setup at a specific point in time. For example, a tag can show which project stage or version an object belongs to.
- Resource usage monitoring: Giving tags to resources (such as warehouses or tables) can help you accurately monitor resource usage. For example, tags can be used to group data

warehouses by cost centers or business units, making it easier to analyze how resources are used and their efficiency.

Use tags

Query existing tag information

You can use the following commands to check the structure and content of the `pg_tag` and `pg_tag_description` tables:

```
\d+ pg_tag;  -- Views the detailed structure of the pg_tag table.
\d+ pg_tag_description;  -- Views the detailed structure of the pg_tag_description
table.
SELECT * FROM pg_tag;  -- Gets all tag information.
SELECT * FROM pg_tag_description;  -- Gets all tag description information.
```

You can use the following commands to query the created tags and descriptions:

```
SELECT tagname, tagowner, allowed_values FROM pg_tag ORDER BY 1;  -- Gets all tag
names, owners, and allowed values.
SELECT count(*) FROM pg_tag_description;  -- Gets the total count of tag descriptions.
```

You can also use the following views to quickly query tag information in the database system:

- `database_tag_descriptions`: Queries tag information for database objects.
- `user_tag_descriptions`: Queries tag information for user (role) objects.
- `warehouse_tag_descriptions`: Queries tag information for warehouses.
- `schema_tag_descriptions`: Queries tag information for schemas.
- `relation_tag_descriptions`: Queries tag information for relation objects such as tables, and views.

Example queries:

```
SELECT * FROM database_tag_descriptions;
SELECT * FROM user_tag_descriptions;
SELECT * FROM warehouse_tag_descriptions;
SELECT * FROM tablespace_tag_descriptions;
SELECT * FROM schema_tag_descriptions;
```

(continues on next page)

(continued from previous page)

```
SELECT * FROM relation_tag_descriptions;
```

Create tags

You can create a tag using the `CREATE TAG` statement and specify allowed values. The syntax is:

```
CREATE TAG [ IF NOT EXISTS ] <tag_name>
CREATE TAG [ IF NOT EXISTS ] <tag_name> [ ALLOWED_VALUES '<val_1>' [ , '<val_2>' [ , .
..] ] ]
```

In the above statements, `ALLOWED_VALUES` can define up to 300 allowed values. By default, any string (including empty strings) is allowed, and each string can be up to 256 characters long.

Examples:

```
CREATE TAG tag_test_a;  -- Creates a tag named tag_test_a.
CREATE TAG IF NOT EXISTS tag_test_b;  -- Creates tag_test_b if it does not exist.
CREATE TAG tag_test_c ALLOWED_VALUES '123';  -- Creates a tag_test_c tag with the
allowed value '123'.
CREATE TAG tag_test_d ALLOWED_VALUES '123', '456', ' ';  -- Creates a tag_test_d tag
with multiple allowed values.
CREATE TAG IF NOT EXISTS tag_test_e ALLOWED_VALUES '123', 'val1';  -- Creates a tag_
test_e tag if it does not exist.
```

Delete tags

Use the `DROP TAG` statement to delete a tag. The syntax is:

```
DROP TAG [ IF EXISTS ] <tag_name>;
```

Before deleting a tag, the system checks whether the tag is referenced by any database object. If it is, an error is thrown. The user who executes the deletion must be a superuser or the owner of the tag.

Examples:

```
DROP TAG tag_test_a;  -- Deletes the tag_test_a tag.
DROP TAG IF EXISTS tag_test_b;  -- Deletes the tag_test_b tag if it exists.
```

Modify tags

Use the `ALTER TAG` statement to change the tag name or allowed values. The syntax is:

```
ALTER TAG [ IF EXISTS ] <tag_name> RENAME TO <new_name>;
ALTER TAG [ IF EXISTS ] <tag_name> { ADD | DROP } ALLOWED_VALUES '<val_1>' [, '<val_2>
', ...];
ALTER TAG <tag_name> UNSET ALLOWED_VALUES;
```

- `RENAME TO`: Renames the existing tag to a new name.
- `ADD` or `DROP`: Adds or removes allowed values for the tag.
- `UNSET ALLOWED_VALUES`: Resets the allowed values for the tag.

Rename tags

```
ALTER TAG tag_test_a RENAME TO tag_test_a_new;  -- Renames tag_test_a to tag_test_a_
new.
ALTER TAG IF EXISTS tag_test_b RENAME TO tag_test_b_new;  -- If tag_test_b exists,
renames it to tag_test_b_new.
ALTER TAG tag_test_c_new RENAME TO tag_test_c;  -- Renames tag_test_c_new back to tag_
test_c.
ALTER TAG tag_test_d_new RENAME TO tag_test_d;  -- Renames tag_test_d_new to tag_test_
d.
```

Modify allowed values

Example of removing allowed values:

```
ALTER TAG tag_test_a UNSET ALLOWED_VALUES;  -- Removes all allowed values from tag_
test_a.
ALTER TAG tag_test_b UNSET ALLOWED_VALUES;  -- Removes all allowed values from tag_
test_b.
ALTER TAG tag_test_c UNSET ALLOWED_VALUES;  -- Removes all allowed values from tag_
test_c.
```

Example of adding allowed values:

```
ALTER TAG tag_test_a ADD ALLOWED_VALUES 'val1';  -- Adds 'val1' as an allowed value to
tag_test_a.
```

(continues on next page)

(continued from previous page)

```
ALTER TAG tag_test_b ADD ALLOWED_VALUES 'val2', 'val3';  -- Adds multiple allowed
values to tag_test_b.
ALTER TAG IF EXISTS tag_test_c ADD ALLOWED_VALUES ' ';  -- Adds an empty string as an
allowed value to tag_test_c.
```

Assign tags to objects

You can use the `TAG` keyword to assign tags when creating or modifying database objects.

Assign tags to a database

The syntax is:

```
CREATE [ OR REPLACE ] DATABASE <database_name> ... [ TAG ( <tag_name> = '<tag_value>'
[, ...] ) ];
ALTER DATABASE [ IF EXISTS ] <database_name> TAG ( <tag_name> = '<tag_value>' [, ...]
);
ALTER DATABASE [ IF EXISTS ] <database_name> UNSET TAG ( <tag_name> [, <tag_name>, ...
] );
```

You can assign or remove tags for a database object using the `TAG` keyword in the `CREATE DATABASE` and `ALTER DATABASE` statements.

Examples:

```
CREATE DATABASE sales_db TAG ( environment = 'production' );

ALTER DATABASE sales_db TAG ( environment = 'staging' );  -- Updates the environment
tag of the sales_db database to staging.
ALTER DATABASE sales_db UNSET TAG ( environment );  -- Removes the environment tag
from the sales_db database.
```

Assign tags to tables

The syntax is:

```
CREATE [ OR REPLACE ] TABLE <table_name> ... [ TAG ( <tag_name> = '<tag_value>' [, ...
] ) ];
ALTER TABLE [ IF EXISTS ] <table_name> TAG ( <tag_name> = '<tag_value>' [, ...] );
ALTER TABLE [ IF EXISTS ] <table_name> UNSET TAG ( <tag_name> [, <tag_name>, ...] );
```

- You can use `TAG` in the `CREATE TABLE` and `ALTER TABLE` statements to assign or remove tags from table objects.
- A database object can have up to 50 tags, and each tag must be unique.

Example:

```
CREATE TABLE orders (
    order_id SERIAL,
    order_date DATE
) TAG ( priority = 'high' );

ALTER TABLE orders TAG ( priority = 'urgent' );
ALTER TABLE orders UNSET TAG ( priority );
```

Assign tags to users

The syntax is:

```
CREATE [ OR REPLACE ] USER <user_name> ... [ TAG ( <tag_name> = '<tag_value>' [, ...]
) ];
ALTER USER [ IF EXISTS ] <user_name> TAG ( <tag_name> = '<tag_value>' [, ...] );
ALTER USER [ IF EXISTS ] <user_name> UNSET TAG ( <tag_name> [, <tag_name>, ...] );
```

You can use `TAG` in the `CREATE USER` and `ALTER USER` statements to assign or remove tags from user objects.

Example:

```
CREATE USER john_doe TAG ( role = 'admin' );

ALTER USER john_doe TAG ( role = 'super_admin' );
ALTER USER john_doe UNSET TAG ( role );
```

Assign tags to warehouses

The syntax is:

```
CREATE [ OR REPLACE ] WAREHOUSE <warehouse_name> ... [ TAG ( <tag_name> = '<tag_value>
' [, ...] ) ];
ALTER WAREHOUSE [ IF EXISTS ] <warehouse_name> TAG ( <tag_name> = '<tag_value>' [, ...
```

(continues on next page)

(continued from previous page)

```
] );
ALTER WAREHOUSE [ IF EXISTS ] <warehouse_name> UNSET TAG ( <tag_name> [, <tag_name>, .
..] );
```

You can use `TAG` in the `CREATE WAREHOUSE` and `ALTER WAREHOUSE` statements to assign or remove tags from warehouse objects.

For example:

```
CREATE WAREHOUSE analytics_wh TAG ( cost_center = 'analytics' );

ALTER WAREHOUSE analytics_wh TAG ( cost_center = 'finance' );
ALTER WAREHOUSE analytics_wh UNSET TAG ( cost_center );
```

Tag comments

Use `COMMENT ON TAG` to add a comment to a tag. The syntax is:

```
COMMENT ON TAG <tag_name> IS '<comment>';
```

Example:

```
COMMENT ON TAG priority IS 'This tag indicates the urgency level of orders.';
```

System tables related to tags

SynxDB Elastic uses two metadata tables to store tag information and the relationship between database objects and tags: `pg_tag` and `pg_tag_description`.

- `pg_tag`: Stores information about all tags, including the tag's OID, name, owner, and allowed values.
- `pg_tag_description`: Records the relationship between database objects and tags, including the object's database ID, class ID, object ID, sub-object number, tag OID, and tag value.

pg_tag table

The `pg_tag` table is used to store information about all tags, including the tag identifier, name, owner, and allowed values. The table structure is as follows:

Column name	Data type	Description
`OID`	`OID`	Unique identifier for the tag.
`TAGNAME`	`NAMEDATA`	Name of the tag.
`TAGOWNER`	`OID`	The OID of the tag owner (user).
`ALLOWED_VALUES`	`ARRAY[TEXT]`	Allowed values for this tag (if any).

With the `pg_tag` table, the database system can efficiently manage and store all defined tag information, and each tag can have a set of allowed values, making it flexible to use.

pg_tag_description table

The `pg_tag_description` table records the relationships between database objects and tags. Each record represents a tag and its value associated with a specific database object. The structure of the table is as follows:

Column name	Data type	Description
`oid`	`OID`	The unique identifier for the table.
`tddatabaseid`	`OID`	The database ID associated with the tag (set to `0` for globally shared objects).
`tdclassid`	`OID`	The class ID of the database object.
`tdobjid`	`OID`	The `OID` of the database object.
`tagid`	`OID`	The `OID` of the tag, indicating the tag associated with the object.
`tagvalue`	`TEXT`	The specific value of the tag for the object.

The `pg_tag_description` table stores the tag information for each database object along with its values, supporting many-to-many relationships between tags and objects.

Handle global and non-global objects

For globally shared objects (such as users and repositories), the `DBID` field value is `0`. This means that these objects are not part of a specific database and have global properties. For non-global shared objects (such as a specific database object), the `DBID` field stores the OID of the database where the object belongs.

Sub-object number (`OBJSUBID`): Currently, the database system does not support adding labels to

sub-objects (like columns), so `OBJSUBID` is always `0`.

Common errors and tips

- **Duplicate labels**: If you assign duplicate labels to the same object, the database system returns an error message.
- **Permission denied**: If a non-superuser or non-label owner tries to delete or modify a label, the database system returns a permission error.
- **Exceeded label limit**: An object can have a maximum of 50 labels. Adding more than 50 will cause an error.

Using this feature, database administrators can organize and manage database objects more effectively, making the system more flexible and easier to operate.

4.1.8 Data Branching

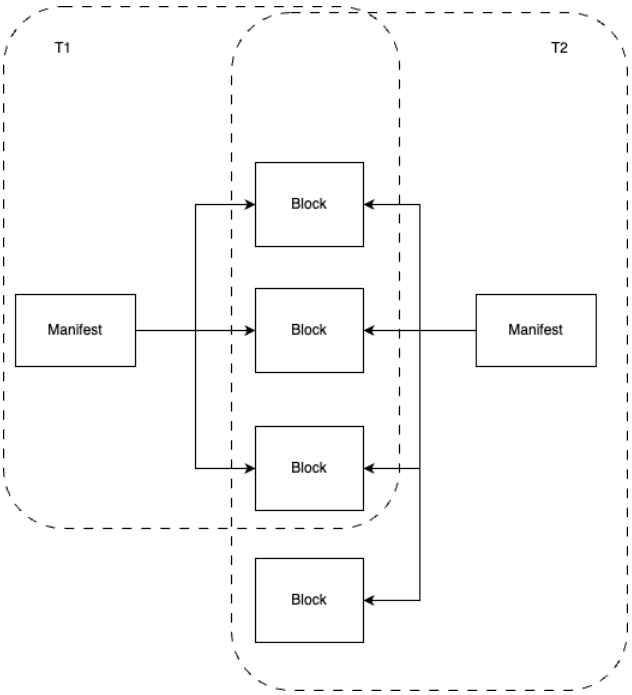
In SynxDB Elastic, data branching is a powerful feature that allows users to create a fully functional copy of an existing table at a very low cost. It works similarly to the "branch" concept in the Git version control system.

What it is: manage data like Git

The data branching feature allows you to quickly create a new table (for example, `t2`) based on an existing one (for example, `t1`). This creation process is highly efficient because it does not initially copy any physical data. Instead, the newly created branch table shares the same set of underlying data blocks with the original table.

When you perform a write operation (such as `INSERT`, `UPDATE`, `DELETE`) on any branch table, the system uses a "Copy-on-Write" mechanism. It only creates new, independent data blocks for the branch where the changes occur, while the original data remains untouched.

The figure above shows `t2` as a branch of `t1`. Initially, they share the same data blocks. When `t2` is written to, it creates its own new data blocks.



What it's for: efficient, isolated, zero-cost data snapshots

Using the data branching feature offers the following core advantages:

- **Instantaneous creation, cost savings**: Creating a branch is almost instantaneous because it avoids large-scale data copying, which significantly saves time and storage space.
- **Safe and isolated experimental environment**: Any operations performed on a branch table (including modifying, deleting data, or even `TRUNCATE`) do not affect the original table or other branches. This provides a perfect sandbox environment for data experiments and development without worrying about damaging production data.
- **Simplified data version management**: You can create different data branches for different tasks (such as daily reports, data cleansing, machine learning training, etc.), making it easy to manage multiple versions of your data.

Use cases

Data branching is ideal for the following scenarios:

- **ETL/ELT development and testing**: You can create a branch from a production data table and then develop and debug your data transformation scripts on that branch. If something goes wrong, you can simply delete the branch without any impact on the production data.

- **Data analysis and scientific exploration**: Data analysts can create a private data branch for themselves. On this branch, they can freely add columns, clean data, and perform various exploratory analyses without interfering with the team's shared original dataset.
- **"What-if" analysis**: Create a branch based on current data, and then simulate the impact of business changes (such as price adjustments, marketing campaigns, etc.) in the branch. This makes complex data forecasting and modeling both safe and efficient.
- **Software development and continuous integration (CI/CD)**: In the application testing process, a database branch can be automatically created for each test task. This ensures that each test runs in a clean, isolated environment, avoiding interference between tests.

How to use

Create a branch

```
BRANCHING new_table_name FROM existing_table_name;
```

Example: Suppose there is a production table named `sales_prod`, and you want to create a branch for quarterly report analysis.

```
BRANCHING sales_q2_report FROM sales_prod;
```

After execution, a new table named `sales_q2_report` will be created. Initially, it shares all data with `sales_prod`. Subsequently, all modifications to `sales_q2_report` will be independent of `sales_prod`.

Considerations

- **Table type limitation**: This feature is only supported for `storageam` type tables.
- **Automatic background management**: SynxDB Elastic automatically tracks the reference status of data blocks in the background. When you use commands like `DROP TABLE`, `TRUNCATE`, or `VACUUM`, the system intelligently determines if a data block is still referenced by other branches and ensures that data is only deleted when there are no more references, thus guaranteeing data safety. Users do not need to worry about this internal mechanism and can operate branch tables just like regular tables.
- **Affected commands**: The `BRANCHING`, `DROP TABLE`, `TRUNCATE TABLE`, and `VACUUM` commands interact with the branch's reference counting system to ensure data consistency

and integrity.

4.2 Choose Table Storage Model

4.2.1 Heap and Append-optimized Table Storage Formats

SynxDB Elastic provides heap and append-optimized storage models. The storage model you choose depends on the type of data you are storing and the type of queries you are running. This section provides an overview of the two storage models and guidelines for choosing the optimal storage model for your data.

Heap storage

By default, SynxDB Elastic uses the same heap storage model as PostgreSQL. Heap table storage works best with OLTP-type workloads where the data is often modified after it is initially loaded. `UPDATE` and `DELETE` operations require storing row-level versioning information to ensure reliable database transaction processing. Heap tables are best suited for smaller tables, such as dimension tables, that are often updated after they are initially loaded.

Append-optimized storage

Append-optimized table storage works best with denormalized fact tables in a data warehouse environment. Denormalized fact tables are typically the largest tables in the system. Fact tables are usually loaded in batches and accessed by read-only queries. Moving large fact tables to an append-optimized storage model eliminates the storage overhead of the per-row update visibility information. This allows for a leaner and easier-to-optimize page structure. The storage model of append-optimized tables is optimized for bulk data loading. Single row `INSERT` statements are not recommended.

To create a table with specified storage options

Row-oriented heap tables are the default storage type.

```
CREATE TABLE foo (a int, b text);
```

Use the `WITH` clause of the `CREATE TABLE` command to declare the table storage options. The default is to create the table as a regular row-oriented heap-storage table. For example, to create an append-optimized table with no compression:

```
CREATE TABLE bar (a int, b text)
    WITH (appendoptimized=true);
```

> **Note**
>
> You use the `appendoptimized=value` syntax to specify the append-optimized table storage type. `appendoptimized` is a thin alias for the `appendonly` legacy storage option. SynxDB Elastic stores `appendonly` in the catalog, and displays the same when listing storage options for append-optimized tables.

`UPDATE` and `DELETE` are not allowed on append-optimized tables in a repeatable read or serializable transaction and will cause the transaction to end prematurely.

Choose row or column-oriented storage

SynxDB Elastic provides a choice of storage orientation models: row, column, or a combination of both. This section provides general guidelines for choosing the optimal storage orientation for a table. Evaluate performance using your own data and query workloads.

- **Row-oriented storage:** good for OLTP workloads with many iterative transactions and where many columns of a single row are needed at once, making retrieval efficient.
- **Column-oriented storage:** good for data warehouse workloads with aggregations of data computed over a small number of columns or for single columns that require regular updates without modifying other column data.

For most general-purpose or mixed workloads, row-oriented storage offers the best combination of flexibility and performance. However, there are use cases where a column-oriented storage model provides more efficient I/O and storage. Consider the following requirements when deciding on the storage orientation model for a table:

- **Updates of table data:** If you load and update table data frequently, choose a row-oriented heap table. Column-oriented table storage is only available on append-optimized tables.
- **Frequent INSERTs:** If rows are frequently inserted into the table, consider a row-oriented model. Column-oriented tables are not optimized for write operations because column values for a row must be written to different places on disk.
- **Number of columns requested in queries:** If you typically request all or the majority of columns in the `SELECT` list or `WHERE` clause of your queries, consider a row-oriented model. Column-oriented tables are best suited to queries that aggregate many values of a single column where the `WHERE` or `HAVING` predicate is also on the aggregate column. For example:

```
SELECT SUM(salary) ...
```

```
SELECT AVG(salary) ... WHERE salary > 10000
```

Or where the `WHERE` predicate is on a single column and returns a relatively small number of rows. For example:

```
SELECT salary, dept ... WHERE state='CA'
```

- **Number of columns in the table:** Row-oriented storage is more efficient when many columns are required at the same time, or when the row size of a table is relatively small. Column-oriented tables can offer better query performance on tables with many columns where you access a small subset of columns in your queries.
- **Compression:** Column data has the same data type, so storage size optimizations are available in column-oriented data that are not available in row-oriented data. For example, many compression schemes use the similarity of adjacent data to compress. However, the greater adjacent compression achieved, the more difficult random access can become, as data must be uncompressed to be read.

To create a column-oriented table

The `WITH` clause of the `CREATE TABLE` command specifies the table's storage options. The default is a row-oriented heap table. Tables that use column-oriented storage must be append-optimized tables. For example, to create a column-oriented table:

```
CREATE TABLE bar (a int, b text)
    WITH (appendoptimized=true, orientation=column);
```

4.3 Perform SQL Queries

4.3.1 Table Join Queries

In SynxDB Elastic, the `JOIN` clause is used to combine rows from two or more tables, based on a related column between them. The `JOIN` clause is part of the `FROM` clause in a `SELECT` statement.

The syntax for the `JOIN` clause is as follows:

```
table1_name join_type table2_name [join_condition]
```

Where:

- `table1_name`, `table2_name`: The names of the tables to be joined.
- `join_type`: The type of join, which can be one of the following:
  - `[INNER] JOIN`
  - `LEFT [OUTER] JOIN`
  - `RIGHT [OUTER] JOIN`
  - `FULL [OUTER] JOIN`
  - `CROSS JOIN`
  - `NATURAL JOIN`
- `join_condition`: An optional join condition that specifies how to match rows between the two tables. It can be one of the following:
  - `ON <join_condition>`
  - `USING ( <join_column> [, ...] )`
  - `LATERAL`

Note

The ORCA optimizer automatically chooses between `Merge Join` or `Hash Join` for `FULL JOIN` queries based on cost estimation. You do not need to manually specify the `JOIN` type.

Join types

SynxDB Elastic supports the following join types:

INNER JOIN

`INNER JOIN` returns the intersection of rows from both tables that satisfy the join condition. In other words, it returns only the matching rows from both tables. If you omit the `INNER` keyword before `JOIN`, it defaults to an `INNER JOIN`.

```
SELECT *
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

LEFT OUTER JOIN

`LEFT OUTER JOIN` (or simply `LEFT JOIN`) returns all rows from the left table, along with the matched rows from the right table. If there is no match for a row from the left table in the right table, the columns of the right table will be filled with `NULL` values.

```
SELECT *
FROM table1
LEFT OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

RIGHT OUTER JOIN

`RIGHT OUTER JOIN` (or simply `RIGHT JOIN`) is the opposite of `LEFT OUTER JOIN`. It returns all rows from the right table, along with the matched rows from the left table. If there is no match for a row from the right table in the left table, the columns of the left table will be filled with NULL values.

```
SELECT *
FROM table1
RIGHT OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

> **Note**
>
> Starting from v2.0.0, `Hash Right Join` queries can also trigger Dynamic Partition Elimination (DPE) when partition pruning conditions are met, which reduces partition scanning and improves performance.

FULL OUTER JOIN

`FULL OUTER JOIN` (or simply `FULL JOIN`) returns all rows from both the left and right tables. For rows from the right table that have no matching rows in the left table, the columns of the left table will be filled with NULL values. For rows from the left table that have no matching rows in the right table, the columns of the right table will be filled with NULL values.

```
SELECT *
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

CROSS JOIN

`CROSS JOIN` returns the Cartesian product of the two tables. It combines each row from the left table with each row from the right table. Without a `WHERE` clause to filter the results, this produces a table with a number of rows equal to the number of rows in the left table multiplied by the number of rows in the right table.

```
SELECT *
FROM table1
CROSS JOIN table2;
```

NATURAL JOIN

The `NATURAL` clause is a shorthand for the `USING` clause. It can be used when all columns with the same name in both tables are used for the join. If the two tables have no column names in common, `NATURAL JOIN` is equivalent to `CROSS JOIN`. You should use `NATURAL JOIN` with caution, as its reliance on column names can lead to unexpected results.

```
SELECT *
FROM table1
```

(continues on next page)

(continued from previous page)

```
NATURAL JOIN table2;
```

Join conditions

Join conditions specify how to match rows between two tables. You can use the following join conditions:

ON clause

The `ON` clause specifies a boolean expression that determines which rows from the two tables are considered a match. It is similar to a `WHERE` clause but applies only to the `JOIN` operation.

```
SELECT *
FROM table1
JOIN table2
ON table1.column_name = table2.column_name;
```

USING clause

The `USING` clause is a shorthand for the `ON` clause, used when the two tables have one or more columns with the same name. It specifies the common columns to be used for the join. The result of a `USING` clause is that the output contains only one of each pair of equivalent columns, instead of both.

```
SELECT *
FROM table1
JOIN table2
USING (column_name);
```

LATERAL

The `LATERAL` keyword can be placed before a subquery `FROM` item. This allows the subquery to reference columns of `FROM` items that appear before it in the `FROM` list. (Without `LATERAL`, SynxDB Elastic would evaluate each subquery independently, so cross-referencing other `FROM` items would not be possible.)

Examples

Suppose we have two tables: `customers` and `orders`.

`customers` table:

```
customer_id | customer_name
------------+---------------
1           | John Doe
2           | Jane Smith
3           | Bob Johnson
```

`orders` table:

```
order_id | customer_id | order_date
---------+-------------+------------
1        | 1           | 2023-01-15
2        | 2           | 2023-02-20
3        | 1           | 2023-03-10
4        | 4           | 2024-05-01
```

Here are some examples using different `JOIN` types:

INNER JOIN example

This query returns all customers and their orders, including only rows with matching customer IDs.

```
SELECT customers.customer_name, orders.order_id
FROM customers
INNER JOIN orders
ON customers.customer_id = orders.customer_id;
```

Result:

```
customer_name | order_id
--------------+----------
John Doe      | 1
Jane Smith    | 2
John Doe      | 3
```

LEFT OUTER JOIN example

This query returns all customers and their orders. Customer information is returned even if a customer has no orders.

```
SELECT customers.customer_name, orders.order_id
FROM customers
LEFT OUTER JOIN orders
ON customers.customer_id = orders.customer_id;
```

Result:

```
customer_name | order_id
--------------+----------
John Doe      | 1
Jane Smith    | 2
Bob Johnson   | NULL
John Doe      | 3
```

RIGHT OUTER JOIN example

This query returns all orders and the customers who placed them. Order information is returned even if an order has no associated customer. In this example, the order with `order_id` 4 has no associated customer.

```
SELECT customers.customer_name, orders.order_id
FROM customers
RIGHT OUTER JOIN orders
ON customers.customer_id = orders.customer_id;
```

Result:

```
customer_name | order_id
--------------+----------
John Doe      | 1
Jane Smith    | 2
John Doe      | 3
NULL          | 4
```

FULL OUTER JOIN example

This query returns all customers and orders. If a customer has no orders or an order has no customer, the information for that customer or order is still included in the result.

```
SELECT customers.customer_name, orders.order_id
FROM customers
FULL OUTER JOIN orders
ON customers.customer_id = orders.customer_id;
```

Result:

```
customer_name | order_id
--------------+----------
John Doe      | 1
Jane Smith    | 2
Bob Johnson   | NULL
John Doe      | 3
NULL          | 4
```

CROSS JOIN example

This query returns the Cartesian product of the `customers` and `orders` tables.

```
SELECT customers.customer_name, orders.order_id
FROM customers
CROSS JOIN orders;
```

Result (partially shown, there are 3 * 4 = 12 rows in total):

```
customer_name | order_id
--------------+----------
John Doe      | 1
John Doe      | 2
John Doe      | 3
John Doe      | 4
Jane Smith    | 1
Jane Smith    | 2
...
```

4.3.2 Cross-Cluster Federated Query

SynxDB Elastic has supported cross-cluster federated queries, allowing you to query data across multiple PostgreSQL database clusters. This feature enables cross-cluster data pushdown, allowing queries to be executed locally in multiple clusters, and then the results are aggregated. The pushdown capability effectively reduces data transfer and improves query performance and processing efficiency.

This feature is implemented through the MPP foreign data wrapper `postgres_fdw`, which allows remote PostgreSQL databases to be treated as shards in SynxDB Elastic, enabling data sharing and federated queries across homogenous systems. Each remote PostgreSQL table can be viewed as an external table, and multiple remote tables can be added to SynxDB Elastic for operations. In this way, you can access and query data from other clusters within one cluster without complex data import or migration operations. This provides great convenience for scenarios that involve handling similar data across multiple clusters.

User scenarios

- **Data integration and analysis**: In some companies, data is often spread across multiple database clusters of the same type. This feature helps companies integrate data across clusters for unified analysis and processing. The ability to perform cross-cluster federated queries allows businesses to quickly merge data from different units and generate unified reports and analysis results. This integration reduces the complexity of manual data importing and processing, while improving the real-time aspect of data analysis.
- **Simplified data management**: The cross-cluster query feature reduces the need for duplicating data across databases, enabling data in each cluster to be directly involved in queries, avoiding the maintenance of redundant data. By treating remote databases as external shards, you can easily manage and access these shards without worrying about data synchronization. At the same time, this also reduces the potential risks and maintenance costs associated with cross-cluster data synchronization.
- **Efficient distributed querying**: By pushing down queries to reduce network traffic and bottlenecks caused by centralized processing, this feature is especially suited for large-scale distributed data analyses. When handling big data, distributed querying can significantly reduce the computing and communication load, enhancing the system's overall processing capacity. Especially when performing complex analytical calculations across multiple clusters, pushdown queries effectively utilize the computing resources of each cluster, improving

processing efficiency.

Usages

This section explains how to perform cross-cluster federated queries using the `postgres_fdw` extension.

Prerequisites

- Ensure that all clusters involved in the federated query can communicate with each other. This includes network connectivity and secure access configurations to make sure data can be transmitted smoothly between the clusters without restrictions.

Step 1. Create a foreign data wrapper

1. Load the `postgres_fdw` foreign data wrapper in SynxDB Elastic.

```
CREATE EXTENSION postgres_fdw;
```

2. Create a remote server object, for example, `testserver`.

```
CREATE SERVER testserver FOREIGN DATA WRAPPER postgres_fdw (host '<remote_server_
IP>', dbname '<remote_database_name>', port '<remote_port_number>');
```

You can create multiple server objects as needed, for example, creating `mpps1`, `mpps2`, and `mpps3` for the remote databases `fdw1`, `fdw2`, and `fdw3`:

```
CREATE SERVER mpps1 FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'xxx.xxx.xxx.
xxx', dbname 'fdw1', port '7000');
CREATE SERVER mpps2 FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'xxx.xxx.xxx.
xxx', dbname 'fdw2', port '7000');
CREATE SERVER mpps3 FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'xxx.xxx.xxx.
xxx', dbname 'fdw3', port '7000');
```

These commands allow you to define different server objects for different remote databases, making it easier to create external tables and perform queries later.

Step 2. Create a user mapping

Create a user mapping for each server. You need to fill in the actual database username and password in the `OPTIONS` section.

```
CREATE USER MAPPING FOR CURRENT_USER SERVER mpps1 OPTIONS(user 'postgres', password
'xxx');
CREATE USER MAPPING FOR CURRENT_USER SERVER mpps2 OPTIONS (user 'postgres', password
'xxx');
CREATE USER MAPPING FOR CURRENT_USER SERVER mpps3 OPTIONS (user 'postgres', password
'xxx');
```

User mapping links the current database user with the remote database user, allowing access to remote data when running queries. Depending on your needs, you can also specify different access permissions for each server.

Step 3. Create a foreign table and add shards

Create a foreign table `fs1`, and add the table `t1` from the remote server `mpps1` as a foreign shard:

```
CREATE FOREIGN TABLE fs1 (
    a int,
    b text
) SERVER mpps1 OPTIONS (schema_name 'public', table_name 't1', mpp_execute 'all
segments');

ADD FOREIGN SEGMENT FROM SERVER mpps1 INTO fs1;
```

With these commands, you can add one or more tables from remote databases as foreign shards to an external table. The data from these shards automatically participates in queries. This allows you to access and manage remote shard data just like you would with local tables.

Step 4. Perform Cross-Cluster Federated Query

Single table query

For single table queries, you can directly query the foreign table `fs1`. The database system distributes the query to the remote cluster for execution.

1. Check the query plan.

```
EXPLAIN (COSTS OFF) SELECT * FROM fs1;
```

The execution plan shows `Foreign Scan on fs1`, which indicates that the query operation has been distributed to the remote server for execution.

```
          QUERY PLAN
---
Gather Motion 1:1  (slice1; segments: 1)
   ->  Foreign Scan on fs1
Optimizer: Postgres query optimizer
(3 rows)
```

2. Run the query.

```
SELECT * FROM fs1;
```

The returned data is from the remote table `t1`.

```
a |  b
---+------
1 | fdw1
(1 row)
```

From the output, you can see that the data is only from the `mpps1` server.

3. Add other remote servers as shards.

```
ADD FOREIGN SEGMENT FROM SERVER mpps2 INTO fs1;
ADD FOREIGN SEGMENT FROM SERVER mpps3 INTO fs1;
```

4. Check the query plan again.

```
EXPLAIN (COSTS OFF) SELECT * FROM fs1;
```

```
            QUERY PLAN
---
Gather Motion 3:1  (slice1; segments: 3)
   ->  Foreign Scan on fs1
Optimizer: Postgres query optimizer
```

(continues on next page)

(continued from previous page)

```
(3 rows)
```

5. Run the query again.

```
SELECT * FROM fs1;
```

Returned result:

```
a |  b
---+------
1 | fdw2
1 | fdw1
1 | fdw3
(3 rows)
```

The result shows that the newly added shards `mpps2` and `mpps3` have successfully participated in the query.

Multi-table join

For multi-table join operations, the optimizer automatically generates an appropriate execution plan based on the number of segments in the tables.

1. Create the foreign table `fs2`.

```
CREATE FOREIGN TABLE fs2 (
   a int,
   b text
   )
SERVER mpps1
OPTIONS (schema_name 'public', table_name 't2', mpp_execute 'all segments');
```

2. Add segments.

```
ADD FOREIGN SEGMENT FROM SERVER mpps1 INTO fs2;
ADD FOREIGN SEGMENT FROM SERVER mpps2 INTO fs2;
ADD FOREIGN SEGMENT FROM SERVER mpps3 INTO fs2;
```

3. Execute the join query between the two tables.

```
EXPLAIN (COSTS OFF) SELECT * FROM fs1, fs2 WHERE fs1.a = fs2.a;
```

The query plan shows that the optimizer has chosen `Hash Join`, and the data from both tables is redistributed based on key values, allowing the join operation to be performed across different nodes.

```
                      QUERY PLAN
---
Gather Motion 3:1  (slice1; segments: 3)
   ->  Hash Join
         Hash Cond: (fs1.a = fs2.a)
         ->  Redistribute Motion 3:3  (slice2; segments: 3)
               Hash Key: fs1.a
               ->  Foreign Scan on fs1
         ->  Hash
               ->  Redistribute Motion 3:3  (slice3; segments: 3)
                     Hash Key: fs2.a
                     ->  Foreign Scan on fs2
Optimizer: Postgres query optimizer
(11 rows)
```

4. Run the actual join query.

```
SELECT * FROM fs1, fs2 WHERE fs1.a = fs2.a;
```

```
a |  b   | a |  b
---+------+---+------
1 | fdw1 | 1 | fdw2
1 | fdw1 | 1 | fdw1
1 | fdw1 | 1 | fdw3
1 | fdw2 | 1 | fdw2
1 | fdw2 | 1 | fdw1
1 | fdw2 | 1 | fdw3
1 | fdw3 | 1 | fdw2
1 | fdw3 | 1 | fdw1
1 | fdw3 | 1 | fdw3
(9 rows)
```

The returned results show that all matching rows between `fs1` and `fs2` are successfully joined.

Join pushdown with gp_foreign_server condition

To further optimize query performance, you can add the `gp_foreign_server` condition in the join to enable join pushdown.

1. Check the query plan.

```
EXPLAIN (COSTS OFF) SELECT * FROM fs1, fs2 WHERE fs1.a = fs2.a AND fs1.gp_foreign_
server = fs2.gp_foreign_server;
```

   The query plan shows that the `Foreign Scan` operation has pushed the join down to the remote server, reducing the burden on local computing.

```
            QUERY PLAN
---
Gather Motion 3:1  (slice1; segments: 3)
   ->  Foreign Scan
         Relations: (fs1) INNER JOIN (fs2)
Optimizer: Postgres query optimizer
(4 rows)
```

2. Run the query.

```
SELECT * FROM fs1, fs2 WHERE fs1.a = fs2.a AND fs1.gp_foreign_server = fs2.gp_
foreign_server;
```

   The returned result:

```
a |  b   | a |  b
---+------+---+------
1 | fdw3 | 1 | fdw3
1 | fdw1 | 1 | fdw1
1 | fdw2 | 1 | fdw2
(3 rows)
```

   By adding the `gp_foreign_server` condition, the query is pushed down to the remote server, making fewer matching rows and greatly improving query efficiency.

Aggregate pushdown

Aggregate queries can be pushed down to the remote server to greatly reduce data transfer, improving query efficiency.

1. Run an aggregate query.

```
SELECT count(*) FROM fs1, fs2 WHERE fs1.a = fs2.a;
```

   The returned result:

```
count
---
   9
(1 row)
```

2. Check the query plan.

```
EXPLAIN (COSTS OFF) SELECT count(*) FROM fs1, fs2 WHERE fs1.a = fs2.a AND fs1.gp_
foreign_server = fs2.gp_foreign_server;
```

   The query plan shows that the aggregate operation has been pushed down to the remote server, reducing the aggregation load on the local node.

```
                       QUERY PLAN
---
Finalize Aggregate
   ->  Gather Motion 3:1  (slice1; segments: 3)
         ->  Foreign Scan
               Relations: Aggregate on ((fs1) INNER JOIN (fs2))
Optimizer: Postgres query optimizer

(5 rows)
```

   By pushing the aggregation operation to the remote server, the database system can leverage the remote server's computing power to perform part of the aggregation work, thus improving overall query performance.

4.4 Advanced Data Analytics

4.4.1 Directory Tables

SynxDB Elastic supports directory tables to uniformly manage unstructured data on object storage.

Large-scale AI applications need to handle unstructured, multi-modal datasets. Therefore, AI application developers must continually prepare a large amount high-quality unstructured data, train large models through repeated iterations, and build rich knowledge bases. This creates technical challenges in managing and processing unstructured data.

To address these challenges, SynxDB Elastic introduces directory tables for managing multiple types of unstructured data. Developers can use simple SQL statements to leverage multiple computing engines for unified data processing and application development.

A directory table stores, manages, and analyzes unstructured data objects. Users can either **upload** local files to a directory table for unified management or directly **map** and manage files that already exist in external object storage, to efficiently manage unstructured data.

- **Upload mode**: When an unstructured data file is imported into a directory table, a record (that is, the file’ s metadata) is created in the directory table, and the file itself is loaded into the object storage managed by SynxDB Elastic.
- **Map mode**: This mode directly scans and identifies files in a user-specified external object storage path and synchronizes their metadata to the directory table. The file entities remain in their original object storage location and are managed by the user.

Instructions

Create a directory table

You can create a directory table in a tablespace on local storage or in a tablespace on external storage (such as object storage services or HDFS).

Create in local storage

The syntax for creating a directory table in local storage is as follows. You need to replace `<table_name>` and `<tablespace_name>` with the actual table name and tablespace name.

```
-- Method 1: Does not specify a tablespace, which means creating the directory table
in the existing default tablespace.
```

(continues on next page)

(continued from previous page)

```
CREATE DIRECTORY TABLE <table_name>;

-- Method 2: First creates a tablespace, and then creates the directory table in that
tablespace.
CREATE TABLESPACE <tablespace_name>
       LOCATION '<tablespace_path>';

CREATE DIRECTORY TABLE <table_name>
       TABLESPACE <tablespace_name>;
```

Create in external storage

To create a directory table in external storage, you first need to create a tablespace in the external storage. You must provide the connection information required to access the external storage server, including the server's IP address, protocol, and access keys. The following examples show how to create a directory table on a major cloud provider like Amazon S3 and HDFS.

1. Create a server object and define the connection method for the external data source. SynxDB Elastic supports multiple storage protocols, including S3 object storage and HDFS. The following examples create server objects named `oss_server` and `hdfs_server` on Amazon S3 and HDFS, respectively.

   - Amazon S3:

   ```
   CREATE STORAGE SERVER oss_server OPTIONS(protocol 's3', prefix '<path_prefix>
   ', endpoint '<endpoint_address>', https 'true', virtual_host 'false');
   ```

   - HDFS:

   ```
   CREATE STORAGE SERVER hdfs_server OPTIONS(port '<port_number>', protocol 'hdfs
   ', namenode '<HDFS_node_IP:port>', https 'false');
   ```

   The parameters in the commands above are as follows:

   - `protocol`: The protocol used to connect to the external data source. In the examples above, `'s3'` indicates the protocol for Amazon S3 Object Storage, and `'hdfs'` indicates the HDFS protocol.

   - `prefix`: Sets the path prefix for accessing object storage. When set, all operations are

confined to this specific path, for example, `prefix '/rose-oss-test4/usf1'`. This is typically used to organize and isolate data within the same storage bucket.

- `endpoint`: Specifies the network address of the external object storage service. For example, `'s3.us-west-2.amazonaws.com'` is a specific regional endpoint for Amazon S3 services. SynxDB Elastic can access external data through this endpoint.
- `https`: Specifies whether to connect to the object storage service via HTTPS. In this command, `'false'` means using an unencrypted HTTP connection. This setting may be influenced by data transmission security requirements; using HTTPS is generally recommended to ensure data security.
- `virtual_host`: Determines whether to use virtual host style for bucket access. `'false'` means not using virtual host style bucket access (that is, the bucket name is not included in the URL). This option usually depends on the URL format supported by the storage service provider.
- `namenode`: The IP address of the HDFS node. You need to replace `<HDFS_node_IP:port>` with the actual IP address and port number, such as `'192.168.51.106:8020'`.
- `port`: The port number of the HDFS node. You need to replace `<port_number>` with the actual port number, for example, `8020`.

2. Create a user mapping to provide the current user with the authentication information needed to access these external servers.

   - Amazon S3:

   ```
   CREATE STORAGE USER MAPPING FOR CURRENT_USER STORAGE SERVER oss_server OPTIONS
   (accesskey '<aws_access_key_id>', secretkey '<aws_secret_access_key>');
   ```

   - HDFS:

   ```
   CREATE STORAGE USER MAPPING FOR CURRENT_USER STORAGE SERVER hdfs_server
   OPTIONS (auth_method 'simple');
   ```

   The parameters in the commands above are as follows:

   - `accesskey` and `secretkey`: These parameters provide the necessary authentication information. `'accesskey'` and `'secretkey'` are like a username and password used

to access the object storage service.

- `auth_method`: Indicates the authentication mode for accessing HDFS. `simple` means simple authentication mode, and `kerberos` means using Kerberos authentication mode.

3. Create a tablespace on the external server. These tablespaces are specifically linked to the previously defined external servers, and the `location` option of the tablespace points to a specific path on the external storage. The following examples create tablespaces `dir_oss` and `dir_hdfs` on Amazon S3 and HDFS, respectively.

   - Amazon S3:

   ```
   CREATE TABLESPACE dir_oss location '<tablespace_path_on_object_storage>'
   SERVER oss_server HANDLER '$libdir/dfs_tablespace, remote_file_handler';

   -- You need to replace <tablespace_path_on_object_storage> with the actual
   path,
   -- for example, /tbs-49560-0-mgq-multi/oss-server-01-17.
   ```

   - HDFS:

   ```
   CREATE TABLESPACE dir_hdfs location '<tablespace_path_on_object_storage>'
   SERVER hdfs_server HANDLER '$libdir/dfs_tablespace, remote_file_handler';

   -- You need to replace <tablespace_path_on_object_storage> with the actual
   path,
   -- for example, /tbs-49560-0-mgq-multi/oss-server-01-17.
   ```

4. Create a directory table in the tablespace. The following statements create directory tables `dir_table_oss` and `dir_table_hdfs` in tablespaces `dir_oss` and `dir_hdfs`, respectively.

   ```
   CREATE DIRECTORY TABLE dir_table_oss TABLESPACE dir_oss;
   CREATE DIRECTORY TABLE dir_table_hdfs TABLESPACE dir_hdfs;
   ```

View field information of a directory table

```
\dY   -- Lists all current directory tables.
\d <directory_table>   -- Views the field information of a specific directory table.
```

The fields of a directory table are typically as follows:

Field name	Data type	Notes
RELATIVE_PAT	TEXT	
SIZE	NUMBER	
LAST_MODIFIE	TIMESTAMP_LT	
MD5	HEX	
TAG	TEXT	User-defined tags. Can be used to mark data lineage, uploading department/team, classification. “k1=v1, k2=v2” .

Field descriptions

Add files to a directory table

Directory tables support two ways to manage files: uploading new files or mapping files that already exist in external object storage.

Upload files

In upload mode, the file entity is copied to a storage space uniformly managed by SynxDB Elastic, while its metadata is recorded in the directory table.

The syntax for uploading files from a local path to the database object storage is as follows:

```
\COPY BINARY '<directory_table_name>' FROM '<local_path_to_file>' '<relative_path>';
COPY BINARY '<directory_table_name>' FROM '<local_path_to_file>' '<relative_path>';  -
- The leading \ can be omitted.

-- <directory_table_name> is the name of the directory table.
-- <local_path_to_file> is the local path of the file to be uploaded.
-- <relative_path> is the target path in local or object storage.
-- The file will be uploaded to this path.
```

> **Tip**
>
> It is recommended to use the subdirectory capability of `<path>` to ensure that the uploaded directory path is consistent with the local one, which facilitates file management.

To better manage or track files and data flows, you can also add a `tag` to the upload command to provide additional information or markers:

```
\COPY BINARY '<directory_table_name>' FROM '<local_path_to_file>' '<relative_path>'
WITH tag '<tag_name>';
```

Examples are as follows:

```
-- Uploads a file to the root directory.
\COPY BINARY dir_table_oss FROM '/data/country.data' 'country.data';

-- Uploads a file to a specific path: top_level/second_level.
\COPY BINARY dir_table_oss FROM '/data/region.tbl' 'top_level/second_level/region.tbl
';

-- Uploads a file to the root directory with a tag.
\COPY BINARY dir_table_oss FROM '/data/country1.data' 'country1.data' with tag
'country';

-- Uploads a file to a specific path top_level/second_level with a tag.
\COPY BINARY dir_table_oss FROM '/data/region1.tbl' 'top_level/second_level/region1.
tbl' with tag 'region';
```

Map external object storage files

In map mode, you can directly manage files already stored in external object storage (such as S3, and HDFS) without moving or uploading them. SynxDB Elastic scans the specified external path and synchronizes the file metadata to the directory table.

When creating a directory table, you can use the `WITH LOCATION` clause to directly specify a path on the object storage (for example, a bucket). This way, the directory table is directly associated with the specified external path. Subsequent file mapping operations (such as `ATTACH LOCATION`) will be based on this path.

```
CREATE DIRECTORY TABLE <directory_table_name>
TABLESPACE <tablespace_name>
WITH LOCATION '<oss_bucket_path>';

-- Example
CREATE DIRECTORY TABLE dir_table_with_location
TABLESPACE dir_oss
WITH LOCATION '/test_dirtable';
```

In the example above, `<oss_bucket_path>` is the bucket path on the object storage, for example, `/test_dirtable`. This allows you to use a directory table to manage data in an entire bucket or a specified directory.

Once the directory table is created, you can use the `ALTER DIRECTORY TABLE ... ATTACH LOCATION` statement to attach one or more paths (`LOCATION`) from external object storage to the directory table, thereby managing specific files.

The syntax is as follows:

```
ALTER DIRECTORY TABLE <directory_table_name> ATTACH LOCATION '<location_path>';

-- Alternatively, attaches a path and add a uniform tag to all files under that path.
ALTER DIRECTORY TABLE <directory_table_name> ATTACH LOCATION '<location_path>' WITH
TAG '<tag_name>';
```

- `<directory_table_name>`: The name of the directory table.
- `<location_path>`: The path in the external object storage. This path is relative to the `location` specified when creating the tablespace.
- `<tag_name>`: The tag set for all files under this path.

Examples are as follows:

```
-- Maps the path named nation1.
ALTER DIRECTORY TABLE dir_table_oss ATTACH LOCATION 'nation1';

-- Maps the path named nation2 and add the 'random' tag to its files.
ALTER DIRECTORY TABLE dir_table_oss ATTACH LOCATION 'nation2' WITH TAG 'random';
```

When a `LOCATION` is attached, the directory table records the metadata of all files under that

path. You can query, export, or delete these mapped files just as you would with uploaded files. For example, use `SELECT * FROM <directory_table>` to query all files, or use the `remove_file()` function to delete a specific file.

Export files from a directory table to a local path

You can export data from a directory table to your local file system for backup:

```
\COPY BINARY DIRECTORY TABLE '<directory_table_name>' '<relative_path>' TO '<target_
path>'; -- Downloads to the psql machine.

\COPY BINARY DIRECTORY TABLE '<directory_table_name>' '<relative_path>' TO '<target_
path>';  -- Downloads to the Coordinator machine.

-- <directory_table_name> is the name of the directory table.
-- <relative_path> is the relative path in the directory table.
-- <target_path> is the destination path, including the local file path and the target
file name.
```

Example:

```
-- Downloads a file to the local root directory.
\COPY BINARY DIRECTORY TABLE dir_table_oss 'country.data' TO '/data/country.CSV';
```

You can also use the `gpdirtableload` command-line tool to bulk download files to the local file system. For details on command-line parameters, see the `gpdirtableload_usage` section.

Query and use directory table files

Query file metadata within a directory table:

```
-- Uses the directory_table() table function to read file metadata and content.
SELECT relative_path,
       size,
       last_modified,
       md5,
       tag,
       content
       FROM directory_table('<directory_table>');

-- Uses any of the following statements to query data in a directory table.
```

(continues on next page)

(continued from previous page)

```
SELECT * FROM <directory_table>;
SELECT * FROM DIRECTORY_TABLE('<directory_table>');
```

Add transaction locks to a directory table

Directory tables support locking statements to control concurrent access, ensuring data consistency and integrity. By using different lock modes, you can restrict other transactions' access to the table to avoid potential data conflicts.

The syntax for locking a table is as follows:

```
LOCK TABLE <table_name> IN <lock_mode>;
```

Where `<lock_mode>` can be one of the following:

- `ACCESS SHARE MODE`: Allows other transactions to read the table but not modify it. Used for read-only queries.
- `ROW SHARE MODE`: Allows other transactions to read rows of the table but not modify them. Suitable for `SELECT FOR UPDATE`/`FOR SHARE`.
- `ROW EXCLUSIVE MODE`: Allows a transaction to update the table. Other transactions can read but not modify.
- `SHARE UPDATE EXCLUSIVE MODE`: Allows online maintenance operations. Other transactions can read but not modify.
- `SHARE MODE`: Allows other transactions to read the table but not modify it.
- `SHARE ROW EXCLUSIVE MODE`: Allows certain operations, such as creating triggers. Other transactions can read but not modify.
- `EXCLUSIVE`: Allows only concurrent `Access Share` locks. The lock holder can only perform read-only operations on the table. Typically acquired by `REFRESH MATERIALIZED VIEW CONCURRENTLY`.
- `ACCESS EXCLUSIVE MODE`: The highest level of lock, ensuring the holder is the only transaction accessing the table. Acquired by commands like `DROP TABLE`, `TRUNCATE`, and `VACUUM FULL`.

Here is an example of using the `ACCESS SHARE MODE` lock:

```
BEGIN;
LOCK TABLE dir_table1 IN ACCESS SHARE MODE;  -- In this mode, other transactions can
still query dir_table1.
SELECT * FROM dir_table1;  -- Queries dir_table1.
LOCK TABLE dir_table1 IN ACCESS EXCLUSIVE MODE;  -- Requests an exclusive lock; other
transactions cannot access dir_table1.
COMMIT;
```

Here is an example of using the `ACCESS EXCLUSIVE MODE` lock:

```
BEGIN;
LOCK TABLE dir_table1 IN ACCESS EXCLUSIVE MODE;  -- In this mode, other transactions
cannot query or modify dir_table1.

-- Performs some operations.
ROLLBACK;  -- Rolls back the operations and releases the lock.
```

Delete files managed by a directory table

To delete files managed by a directory table, you need administrator privileges:

```
SELECT remove_file('dir_table_oss', 'country.data');

-- This command deletes the file country.data managed by the dir_table_oss table.
```

Drop a directory table

Deletes the specified directory table. After deletion, all files within the table are also deleted. You need administrator privileges to drop a directory table.

```
DROP DIRECTORY TABLE <table_name>;
```

4.4.2 Use pgvector for Vector Similarity Search

pgvector is an open-source plugin for vector similarity search. It supports both exact and approximate nearest neighbor searches, as well as L2 distance, inner product, and cosine distance. For more details, see pgvector/pgvector: Open-source vector similarity search for Postgres.

SynxDB Elastic allows you to use pgvector for data storage, querying, hybrid searches, and more through SQL statements. This document explains how to use pgvector in SynxDB Elastic.

Quick start

Enable the extension (do this once in each database where you want to use it):

```
CREATE EXTENSION vector;
```

Create a vector column with 3 dimensions:

```
CREATE TABLE items (id bigserial PRIMARY KEY, embedding vector(3));
```

Insert vector data:

```
INSERT INTO items (embedding) VALUES ('[1,2,3]'), ('[4,5,6]');
```

Get the nearest neighbors by L2 distance:

```
SELECT * FROM items ORDER BY embedding <-> '[3,1,2]' LIMIT 5;
```

Note: Use <#> for inner product and <=> for cosine distance.

Store data

Create a table with a vector column:

```
CREATE TABLE items (id bigserial PRIMARY KEY, embedding vector(3));
```

Or add a vector column to an existing table:

```
ALTER TABLE items ADD COLUMN embedding vector(3);
```

Insert vectors:

```
INSERT INTO items (embedding) VALUES ('[1,2,3]'), ('[4,5,6]');
```

Insert and update vectors:

```
INSERT INTO items (id, embedding) VALUES (1, '[1,2,3]'), (2, '[4,5,6]')
    ON CONFLICT (id) DO UPDATE SET embedding = EXCLUDED.embedding;
```

Update vectors:

```
UPDATE items SET embedding = '[1,2,3]' WHERE id = 1;
```

Delete vectors:

```
DELETE FROM items WHERE id = 1;
```

Query data

Get the nearest neighbors to a vector:

```
SELECT * FROM items ORDER BY embedding <-> '[3,1,2]' LIMIT 5;
```

The supported distance functions are:

- <->: L2 distance
- <#>: negative inner product
- <=>: cosine distance

Get the nearest neighbors of a row:

```
SELECT * FROM items WHERE id != 1 ORDER BY embedding <-> (SELECT embedding FROM items
WHERE id = 1) LIMIT 5;
```

Get rows within a specific distance range:

```
SELECT * FROM items WHERE embedding <-> '[3,1,2]' < 5;
```

Get the distance:

```
SELECT embedding <-> '[3,1,2]' AS distance FROM items;
```

For inner product, multiply by `-1` (because <#> returns the negative inner product).

```
SELECT (embedding <#> '[3,1,2]') * -1 AS inner_product FROM items;
```

For cosine similarity, use `1` minus the cosine distance.

```
SELECT 1 - (embedding <=> '[3,1,2]') AS cosine_similarity FROM items;
```

Calculate the average of vectors:

```
SELECT AVG(embedding) FROM items;
```

Calculate the average of a group of vectors:

```
SELECT category_id, AVG(embedding) FROM items GROUP BY category_id;
```

Hybrid search

Perform hybrid search using SynxDB Elastic full-text search:

```
SELECT id, content FROM items, plainto_tsquery('hello search') query
    WHERE textsearch @@ query ORDER BY ts_rank_cd(textsearch, query) DESC LIMIT 5;
```

pgvector performance

Use `EXPLAIN ANALYZE` for performance debugging:

```
EXPLAIN ANALYZE SELECT * FROM items ORDER BY embedding <-> '[3,1,2]' LIMIT 5;
```

Exact search

To speed up queries, you can increase the value of the `max_parallel_workers_per_gather` parameter.

```
SET max_parallel_workers_per_gather = 4;
```

If vectors are already normalized to a length of `1` (for example, the OpenAI embeddings), using inner product can provide the best performance.

```
SELECT * FROM items ORDER BY embedding <#> '[3,1,2]' LIMIT 5;
```

These are some guidelines for nearest neighbor search and performance optimization in pgvector. Depending on your needs and data structure, you can adjust and optimize based on these recommendations.

4.4.3 Vectorization Query Computing

When handling large datasets, the vectorization execution engine can greatly improve computing efficiency. By vectorizing data, multiple data elements can be processed at the same time, using parallel computing and SIMD instruction sets to speed up the process. SynxDB Elastic Vectorization (referred to as Vectorization) is a vectorization plugin based on the SynxDB Elastic kernel, designed to optimize query performance.

Enable vectorization

```
SET vector.enable_vectorization = ON;
```

Usage

Vectorization provides two parameters for users: `vector.enable_vectorization` and `vector.max_batch_size`.

Parameter name	Description
`vector.enable_vect`	Controls whether vectorized queries are enabled. It is disabled by default.
`vector.max_batch_s`	The size of the vectorized batch, which controls how many rows the executor processes in one cycle. The range is `[0, 600000]`, and the default value is `16384`.

Enable or disable vectorization queries

You can temporarily enable or disable the vectorization feature in a connection by setting the `vector.enable_vectorization` variable. This setting is effective only for the current connection, and it will reset to the default value after disconnecting.

After uninstalling vectorization, setting the `vector.enable_vectorization` variable will have no effect. When you reinstall it, the `vector.enable_vectorization` will be restored to the default value `on`.

```
SET vector.enable_vectorization TO [on|off];
```

Set vectorization batch size

The batch size of vectorization greatly affects performance. If the value is too small, queries might slow down. If the value is too large, it might increase memory usage without improving performance.

```
SET vector.max_batch_size TO <batch_number>;
```

Verify whether a query is vectorized

You can use `EXPLAIN` to check whether a query is a vectorization query.

- If the first line of the `QUERY PLAN` has a "Vec" label, it means the query uses vectorization.

```
gpadmin=# EXPLAIN SELECT * FROM int8_tbl;
                                    QUERY PLAN

-------------------------------------------------------------------------------
-
Vec Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..431.00 rows=1 width=16)
->  Vec Seq Scan on int8_tbl  (cost=0.00..431.00 rows=1 width=16)
Optimizer: Pivotal Optimizer (GPORCA)
(3 rows)
```

- If the first line of the `QUERY PLAN` does not have a "Vec" label, it means the query does not use vectorization.

```
gpadmin=# EXPLAIN SELECT * FROM int8_tbl;
                                QUERY PLAN
-------------------------------------------------------------------------------
Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..431.00 rows=1 width=16)
->  Seq Scan on int8_tbl  (cost=0.00..431.00 rows=1 width=16)
Optimizer: Pivotal Optimizer (GPORCA)
(3 rows)
```

Supported features

The following table describes the operators, functions, and data types supported by vectorized execution.

Feature	Supported or not	Description
Storage format	Supported	AOCS
Storage format	Not supported	HEAP
Data types	Supported	int2, int4, int8, float8, bool, char, tid, date, time, timestamp, timestamptz, varchar, text, numeric
Data types	Not supported	Custom type
Scan operator	Supported	Scanning AOCS tables, complex filter conditions
Scan operator	Not supported	Non-AOCS tables
Agg operator	Supported	Aggregate functions: min, max, count, sum, avg Aggregation plans: PlanAggregate (simple aggregate), GroupAggregate (sorted aggregate), HashAggregate (hash aggregate)
Agg operator	Not Supported	Aggregate functions: sum(int8), sum(float8), stddev (standard deviation), variance Aggregation plans: MixedAggregate (mixed aggregate)
Limit operator	Supported	All
ForeignScan operator	Supported	All
Result operator	Supported	All
Append operator	Supported	All
Subquery operator	Supported	All
Sequence operator	Supported	All
NestedLoopJoin operator	Supported	Join types: inner join, left join, semi join, anti join
NestedLoopJoin operator	Not supported	Join types: right join, full join, semi-anti join Join conditions: different data types, complex inequality conditions
Material operator	Supported	All
ShareInputScan operator	Supported	All
ForeignScan operator	Supported	All
HashJoin operator	Supported	Join types: inner join, left join, right join, full join, semi join, anti join, semi-anti join
HashJoin operator	Not Supported	Join conditions: different data types, complex inequality conditions
Sort operator	Supported	Sorting order: ascending, descending algorithms: order by, order by limit
Motion operator	Supported	GATHER (sending tuples from multiple senders to one receiver), GATHER_SINGLE (single-node gathering), HASH (simple hash conditions), BROADCAST (broadcast gathering), EXPLICIT (explicit gathering)
Motion operator	Not supported	Hash gathering (complex hash conditions)
Expressions	Supported	case when, is distinct, is not distinct, grouping, groupid, stddev_sample, abs, round, upper, textcat, date_pli, coalesce, substr
Bench	Supported	ClickHouse, TPC-H, TPC-DS, ICW, ICW-ORCA

Single-node threaded execution

By performing threaded parallel computation within the execution node, better utilization of multi-core machine resources can be achieved, reducing query time and improving query performance. The following vectorized operators support threaded acceleration:

- Scan (Filter)
- Join
- Agg
- Sort

Currently, the vectorized query feature supports enabling threaded queries in a single-node deployment. To enable it, the following GUC values need to be configured:

```
set vector.enable_vectorization=on;  -- Enables vectorized query.
set vector.enable_plan_merge=on; -- Enables Pipeline scheduling execution.
set vector.pool_threads=8;  -- Configures the number of execution threads.
```

Vectorized threaded execution relies on the underlying PAX storage file format. Take the `tpcds` PAX table `store_sales` as an example. When threaded execution is not enabled:

```
gpadmin=# set vector.pool_threads=0;
SET
gpadmin=# explain analyze select ss_quantity from store_sales
where ss_quantity between 6 and 10 and ss_list_price between 5 and 5+10;
                                                                  QUERY PLAN


-------------------------------------------------------------------------------------------
----------------------
----------------------------------
Vec Seq Scan on store_sales  (cost=0.00..661957.82 rows=28801 width=4) (actual
time=8804.432..8805.236 rows
=94117 loops=1)
   Filter: ((ss_quantity >= 6) AND (ss_quantity <= 10) AND (ss_list_price >= '5
'::numeric(15,2)) AND (ss_lis
t_price <= '15'::numeric(15,2)))
Planning Time: 0.475 ms
```

(continues on next page)

(continued from previous page)

```
   (slice0)    Executor memory: 114K bytes.
Memory used:  128000kB
Optimizer: Postgres query optimizer
Execution Time: 8809.841 ms
(7 rows)
```

When threaded execution is enabled and the `pool_threads` parameter is set to `8`:

```
gpadmin=# set vector.pool_threads=8;
SET
gpadmin=# explain analyze select ss_quantity from store_sales
where ss_quantity between 6 and 10 and ss_list_price between 5 and 5+10;
                                                                    QUERY PLAN


------------------------------------------------------------------------------------------
----------------------
----------------------------------
```

After enabling threaded queries, the query time is significantly reduced.

4.5 Work with Transactions

Transactions allow you to bundle multiple SQL statements in one all-or-nothing operation.

The following are the SynxDB Elastic SQL transaction commands:

- `BEGIN` or `START TRANSACTION` starts a transaction block.
- `END` or `COMMIT` commits the results of a transaction.
- `ROLLBACK` abandons a transaction without making any changes.
- `SAVEPOINT` marks a place in a transaction and enables partial rollback. You can roll back commands run after a savepoint while maintaining commands run before the savepoint.
- `ROLLBACK TO SAVEPOINT` rolls back a transaction to a savepoint.
- `RELEASE SAVEPOINT` destroys a savepoint within a transaction.

4.5.1 Transaction isolation levels

SynxDB Elastic accepts the standard SQL transaction levels as follows:

- `READ UNCOMMITTED` and `READ COMMITTED` behave like the standard `READ COMMITTED`.
- `REPEATABLE READ` and `SERIALIZABLE` behave like `REPEATABLE READ`.

The following information describes the behavior of the SynxDB Elastic transaction levels.

Read uncommitted and read committed

SynxDB Elastic does not allow any command to see an uncommitted update in another concurrent transaction, so `READ UNCOMMITTED` behaves the same as `READ COMMITTED`. `READ COMMITTED` provides fast, simple, partial transaction isolation. `SELECT`, `UPDATE`, and `DELETE` commands operate on a snapshot of the database taken when the query started.

A `SELECT` query:

- Sees data committed before the query starts.
- Sees updates run within the transaction.
- Does not see uncommitted data outside the transaction.
- Can possibly see changes that concurrent transactions made if the concurrent transaction is

committed after the initial read in its own transaction.

Successive `SELECT` queries in the same transaction can see different data if other concurrent transactions commit changes between the successive queries. `UPDATE` and `DELETE` commands find only rows committed before the commands started.

`READ COMMITTED` transaction isolation allows concurrent transactions to modify or lock a row before `UPDATE` or `DELETE` find the row. `READ COMMITTED` transaction isolation might be inadequate for applications that perform complex queries and updates and require a consistent view of the database.

Repeatable read and serializable

`SERIALIZABLE` transaction isolation, as defined by the SQL standard, ensures that transactions that run concurrently produce the same results as if they were run one after another. If you specify `SERIALIZABLE` SynxDB Elastic falls back to `REPEATABLE READ`. `REPEATABLE READ` transactions prevent dirty reads, non-repeatable reads, and phantom reads without expensive locking, but SynxDB Elastic does not detect all serializability interactions that can occur during concurrent transaction execution. Concurrent transactions should be examined to identify interactions that are not prevented by disallowing concurrent updates of the same data. You can prevent these interactions by using explicit table locks or by requiring the conflicting transactions to update a dummy row introduced to represent the conflict.

With `REPEATABLE READ` transactions, a `SELECT` query:

- Sees a snapshot of the data as of the start of the transaction (not as of the start of the current query within the transaction).
- Sees only data committed before the query starts.
- Sees updates run within the transaction.
- Does not see uncommitted data outside the transaction.
- Does not see changes that concurrent transactions make.
- Successive `SELECT` commands within a single transaction always see the same data.
- `UPDATE`, `DELETE`, `SELECT FOR UPDATE`, and `SELECT FOR SHARE` commands find only rows committed before the command started. If a concurrent transaction has updated, deleted, or locked a target row, the `REPEATABLE READ` transaction waits for the concurrent

transaction to commit or roll back the change. If the concurrent transaction commits the change, the `REPEATABLE READ` transaction rolls back. If the concurrent transaction rolls back its change, the `REPEATABLE READ` transaction can commit its changes.

The default transaction isolation level in SynxDB Elastic is `READ COMMITTED`. To change the isolation level for a transaction, declare the isolation level when you `BEGIN` the transaction or use the `SET TRANSACTION` command after the transaction starts.

4.6 Control Transactional Concurrency

This document introduces the transactional concurrency control in SynxDB Elastic, including:

- *MVCC mechanism*
- *Lock modes*
- *Global Deadlock Detector*

4.6.1 MVCC mechanism

SynxDB Elastic and PostgreSQL do not use locks for concurrency control. Instead, they maintain data consistency through a multi-version model known as Multi-version Concurrency Control (MVCC). MVCC ensures transaction isolation for each database session, allowing each query transaction to see a consistent snapshot of data. This ensures that the data observed by a transaction remains consistent and unaffected by other concurrent transactions.

However, the specific data changes visible to a transaction are influenced by its isolation level. The default isolation level is "READ COMMITTED," which means that a transaction can observe data changes made by other transactions that have already been committed. If the isolation level is set to "REPEATABLE READ," then queries within that transaction will observe the data because it was at the beginning of the transaction and will not see changes made by other transactions in the interim. To specify the isolation level of a transaction, you can use the statement `BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ` to start a transaction with the "REPEATABLE READ" isolation level.

Because MVCC does not use explicit locks for concurrency control, lock contention is minimized and SynxDB Elastic maintains reasonable performance in multi-user environments. Locks acquired for querying (reading) data do not conflict with locks acquired for writing data.

4.6.2 Lock modes

SynxDB Elastic provides multiple lock modes to control concurrent access to data in tables. Most SynxDB Elastic SQL commands automatically acquire the appropriate locks to ensure that referenced tables are not dropped or modified in incompatible ways while a command runs. For applications that cannot adapt easily to MVCC behavior, you can use the `LOCK` command to acquire explicit locks. However, proper use of MVCC generally provides better performance.

Lock mode	SQL commands	Conflicting lock modes
ACCESS SHARE	`SELECT`	ACCESS EXCLUSIVE
ROW SHARE	`SELECT...FOR lock_strength`	EXCLUSIVE, ACCESS EXCLUSIVE
ROW EXCLUSIVE	`INSERT`, `COPY`	SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
SHARE UPDATE EXCLUSIVE	`ANALYZE`	SHARE UPDATE EXCLUSIVE, SHARE, EXCLUSIVE, ACCESS EXCLUSIVE
SHARE		ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
SHARE ROW EXCLUSIVE	/	ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
EXCLUSIVE	`DELETE`, `UPDATE`, `SELECT...FOR lock_strength`, `REFRESH MATERIALIZED VIEW CONCURRENTLY`	ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
ACCESS EXCLUSIVE	`ALTER TABLE`, `DROP TABLE`, `TRUNCATE`, `CLUSTER`, `REFRESH MATERIALIZED VIEW` (without `CONCURRENTLY`) , `VACUUM FULL`	ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE

Note

By default, the Global Deadlock Detector is deactivated, and SynxDB Elastic acquires the more restrictive `EXCLUSIVE` lock (rather than `ROW EXCLUSIVE` in PostgreSQL) for `UPDATE` and `DELETE`.

When the Global Deadlock Detector is enabled, the lock mode for some `DELETE` and `UPDATE` operations on heap tables is `ROW EXCLUSIVE`. See *Global Deadlock Detector*.

4.6.3 Global Deadlock Detector

The SynxDB Elastic Global Deadlock Detector background worker process collects lock information on all segments and uses a directed algorithm to detect the existence of local and global deadlocks. This algorithm allows SynxDB Elastic to relax concurrent update and delete restrictions on heap tables. (SynxDB Elastic still employs table-level locking on AO/CO tables, restricting concurrent `UPDATE`, `DELETE`, and `SELECT...FOR lock_strength` operations.)

By default, the Global Deadlock Detector is deactivated and SynxDB Elastic runs the concurrent

`UPDATE` and `DELETE` operations on a heap table serially. You can activate these concurrent updates and have the Global Deadlock Detector determine when a deadlock exists by setting the parameter `gp_enable_global_deadlock_detector` in the `postgresql.conf` configuration file to `on` and then restarting the database.

When the Global Deadlock Detector is enabled, the background worker process is automatically started on the coordinator host when you start SynxDB Elastic. You configure the interval at which the Global Deadlock Detector collects and analyzes lock waiting data via the `gp_global_deadlock_detector_period` server configuration parameter in the `postgresql.conf` configuration file.

If the Global Deadlock Detector determines that deadlock exists, it breaks the deadlock by cancelling one or more backend processes associated with the youngest transaction(s) involved.

When the Global Deadlock Detector determines a deadlock exists for the following types of transactions, only one of the transactions will succeed. The other transactions will fail with an error indicating that concurrent updates to the same row is not allowed.

- Concurrent transactions on the same row of a heap table where the first transaction is an update operation and a later transaction runs an update or delete and the query plan contains a motion operator.
- Concurrent update transactions on the same row of a hash table that are run by the GPORCA optimizer.

Tip

SynxDB Elastic uses the interval specified in the `deadlock_timeout` server configuration parameter for local deadlock detection. Because the local and global deadlock detection algorithms differ, the cancelled process(es) may differ depending upon which detector (local or global) SynxDB Elastic triggers first.

Tip

If the `lock_timeout` server configuration parameter is turned on and set to a value smaller than `deadlock_timeout` and `gp_global_deadlock_detector_period`, SynxDB Elastic will cancel a statement before it would ever trigger a deadlock check in that session.

To view lock waiting information for all segments, run the `gp_dist_wait_status()` user-defined function. You can use the output of this function to determine which transactions are waiting on locks, which transactions are holding locks, the lock types and mode, the waiter and holder session identifiers, and which segments are running the transactions. Sample output of the `gp_dist_wait_status()` function follows:

```
SELECT * FROM pg_catalog.gp_dist_wait_status();


-[ RECORD 1 ]----+--------------
segid            | 0
waiter_dxid      | 11
holder_dxid      | 12
holdTillEndXact  | t
waiter_lpid      | 31249
holder_lpid      | 31458
waiter_lockmode  | ShareLock
waiter_locktype  | transactionid
waiter_sessionid | 8
holder_sessionid | 9
-[ RECORD 2 ]----+--------------
segid            | 1
waiter_dxid      | 12
holder_dxid      | 11
holdTillEndXact  | t
waiter_lpid      | 31467
holder_lpid      | 31250
waiter_lockmode  | ShareLock
waiter_locktype  | transactionid
waiter_sessionid | 9
holder_sessionid | 8
```

When it cancels a transaction to break a deadlock, the Global Deadlock Detector reports the following error message:

```
ERROR:  canceling statement due to user request: "cancelled by global deadlock
detector"
```

Chapter 5

Optimize Performance

5.1 Optimize Query Performance

5.1.1 Query Performance Overview

Query performance is optimized by dynamically eliminating irrelevant partitions to reduce memory allocation. This mechanism can significantly reduce the amount of data scanned by queries, speed up query execution, and enhance the system's concurrent processing capabilities.

> **Tip**
>
> SynxDB Elastic enables the GPORCA optimizer by default, which extends the query planning and optimization capabilities of the native Postgres optimizer.

Dynamic partition elimination

In SynxDB Elastic, the system dynamically prunes partitions using values that can only be determined at query execution time, thereby improving query processing speed. This feature is called Dynamic Partition Elimination (DPE).

DPE is applicable to the following types of join operations:

- `Hash Inner Join`

- `Hash Left Join`
- `Hash Right Join`

The following conditions must be met to enable DPE:

- The partitioned table must be the outer table of the join.
- The join condition must be an equality condition based on the partitioning key.
- Statistics must be collected on the partitioned table, for example:

```
ANALYZE <root partition>;
```

The `gp_dynamic_partition_pruning` parameter controls whether DPE is enabled and is `ON` by default. This parameter only affects the Postgres optimizer. You can check if DPE is active by using `EXPLAIN` to see if the execution plan contains a `Partition Selector` node.

Memory optimization

SynxDB Elastic dynamically allocates memory based on the characteristics of each operator in a query. It also proactively releases or reallocates memory resources at different stages of the query, leading to more efficient resource utilization and query execution.

5.1.2 Use GPORCA Optimizer

GPORCA Overview

GPORCA is an enhanced optimizer built upon the Postgres query optimizer, augmenting query planning and optimization capabilities. It is highly extensible and achieves more efficient optimization, especially in multi-core architectures. By default, SynxDB Elastic uses GPORCA to generate query execution plans in supported scenarios.

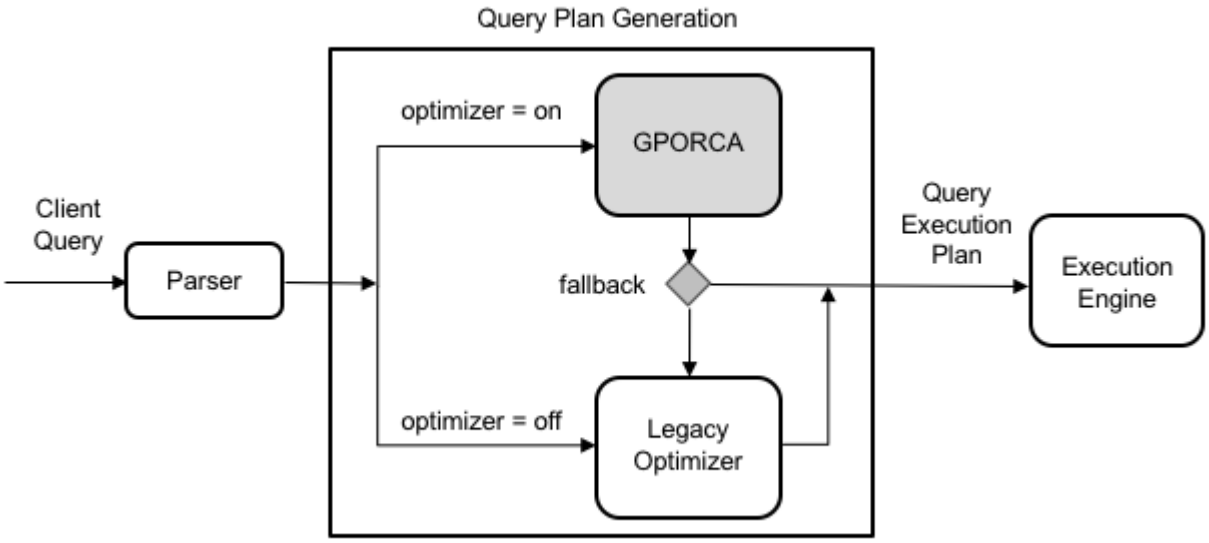
GPORCA significantly enhances the query performance of SynxDB Elastic in the following areas:

- Queries on partitioned tables
- Queries with Common Table Expressions (CTEs)
- Queries with subqueries

In SynxDB Elastic, GPORCA coexists with the Postgres-based optimizer. By default, the system first attempts to use GPORCA. When a query is not supported by GPORCA, the system

automatically falls back to using the Postgres optimizer.

The following diagram illustrates the position of GPORCA in the overall query planning architecture:



Note that all server parameters used to configure the behavior of the Postgres optimizer are ignored when GPORCA is active. These parameters only affect query plan generation when the system falls back to the Postgres optimizer.

Enable or disable GPORCA

You can enable or disable GPORCA using the `optimizer` server configuration parameter.

Although GPORCA is enabled by default, you can flexibly configure the `optimizer` parameter at the system, database, session, or query level to control whether GPORCA is used.

Enable GPORCA for a database

You can use the `ALTER DATABASE` command to enable GPORCA for a specific database. The following example enables GPORCA for the `test_db` database:

```
ALTER DATABASE test_db SET optimizer = on;
```

Enable GPORCA for a session or query

Use the `SET` command to enable GPORCA in the current session. For example, execute the following command after connecting to SynxDB Elastic via `psql`:

```
SET optimizer = on;
```

To enable GPORCA for a single query, run the above `SET` command before executing the query.

Determine which query optimizer is used

When GPORCA is enabled (the default configuration), you can determine whether SynxDB Elastic is actually using GPORCA or has fallen back to the Postgres-based optimizer in several ways.

The most direct method is to examine the `EXPLAIN` plan of the query:

- The optimizer used is indicated at the end of the query plan. For example:
  - If generated by GPORCA, the end shows:

    ```
    Optimizer: GPORCA
    ```

  - If generated by the Postgres optimizer, the end shows:

    ```
    Optimizer: Postgres-based planner
    ```

- If the query plan includes nodes of type `Dynamic <any> Scan` (such as dynamic assertion, dynamic sequence scan, etc.), the plan was generated by GPORCA. The Postgres optimizer does not generate these nodes.
- For queries on partitioned tables, GPORCA's `EXPLAIN` output shows only the number of partitions after pruning, without listing the specific partitions. The Postgres optimizer, on the other hand, lists each partition that is scanned.

In addition to the `EXPLAIN` output, the optimizer type used is also recorded in the logs. If a query is not supported by GPORCA for some reason, the system automatically falls back and outputs relevant information at the `NOTICE` level in the log, explaining the reason for the fallback.

You can also enable the `optimizer_trace_fallback` configuration parameter to display the detailed reasons for GPORCA fallback directly in `psql`.

Examples

The following examples demonstrate the effect of executing queries on a partitioned table when GPORCA is enabled.

The `CREATE TABLE` statement below creates a table that is range-partitioned by date:

```
CREATE TABLE sales (trans_id int, date date,
    amount decimal(9,2), region text)
   PARTITION BY RANGE (date)
      (START (date '2016-01-01')
       INCLUSIVE END (date '2017-01-01')
       EXCLUSIVE EVERY (INTERVAL '1 month'),
   DEFAULT PARTITION outlying_dates );
```

The plan generated by GPORCA only shows the number of selected partitions, without listing their names:

```
->  Partition Selector for sales (dynamic scan id: 1)  (cost=10.00..100.00 rows=50
width=4)
      Partitions selected:  13 (out of 13)
```

If a query on a partitioned table is not supported by GPORCA, the system automatically switches to the Postgres optimizer. The `EXPLAIN` generated by Postgres lists all accessed partitions, as shown below:

```
->  Append  (cost=0.00..0.00 rows=26 width=53)
   ->  Seq Scan on sales2_1_prt_7_2_prt_usa sales2  (cost=0.00..0.00 rows=1 width=53)
   ->  Seq Scan on sales2_1_prt_7_2_prt_asia sales2  (cost=0.00..0.00 rows=1 width=53)
   ...
```

The following example shows the log output when a query falls back to the Postgres optimizer:

Execute the following query:

```
EXPLAIN SELECT * FROM pg_class;
```

The system will use the Postgres optimizer and output a `NOTICE` message in the log, explaining why GPORCA did not handle the query.

```
INFO:  GPORCA failed to produce a plan, falling back to Postgres-based planner
DETAIL:  Falling back to Postgres-based planner because GPORCA does not support the
following feature: Non-default collation
```

GPORCA Features and Enhancements

GPORCA provides enhanced support for certain types of queries and operations:

- Queries on partitioned tables
- Queries with subqueries
- Queries with Common Table Expressions (CTEs)
- DML operation optimization

Enhancements for partitioned table queries

GPORCA introduces the following optimizations for handling queries on partitioned tables:

- Improved partition pruning capabilities.
- The query plan can include the Partition Selector operator.
- The `EXPLAIN` plan no longer enumerates all partitions.

  For queries with static partition pruning (i.e., comparing the partition key with a constant), GPORCA displays the Partition Selector operator in the `EXPLAIN` output, indicating the filter condition and the number of selected partitions. Here is an example:

  ```
  Partition Selector for Part_Table (dynamic scan id: 1)
          Filter: a > 10
          Partitions selected:  1 (out of 3)
  ```

  For queries with dynamic partition pruning (i.e., comparing the partition key with a variable), partition selection is determined during execution, and the `EXPLAIN` output does not list the selected partitions.

- The size of the query plan does not grow with the number of partitions.
- Significantly reduces the risk of out-of-memory errors caused by a large number of partitions.

The following `CREATE TABLE` example creates a range-partitioned table:

```
CREATE TABLE sales(order_id int, item_id int, amount numeric(15,2),
      date date, yr_qtr int)
   PARTITION BY RANGE (yr_qtr) (start (201501) INCLUSIVE end (201504) INCLUSIVE,
   start (201601) INCLUSIVE end (201604) INCLUSIVE,
   start (201701) INCLUSIVE end (201704) INCLUSIVE,
   start (201801) INCLUSIVE end (201804) INCLUSIVE,
   start (201901) INCLUSIVE end (201904) INCLUSIVE,
   start (202001) INCLUSIVE end (202004) INCLUSIVE);
```

GPORCA optimizes the following types of queries on partitioned tables:

- Full table scan: Partitions are not enumerated in the plan.

```
SELECT * FROM sales;
```

- Queries with constant filter conditions: Partition pruning can be performed.

```
SELECT * FROM sales WHERE yr_qtr = 201501;
```

- Range queries: Also trigger partition pruning.

```
SELECT * FROM sales WHERE yr_qtr BETWEEN 201601 AND 201704 ;
```

- Join queries on partitioned tables: The following example joins the dimension table `date_dim` with the fact table `catalog_sales`.

```
SELECT * FROM catalog_sales
    WHERE date_id IN (SELECT id FROM date_dim WHERE month=12);
```

Subquery optimization

GPORCA handles subqueries more efficiently. A subquery is a query nested within an outer query block, such as the `SELECT` in the `WHERE` clause of the following statement:

```
SELECT * FROM part
  WHERE price > (SELECT avg(price) FROM part);
```

GPORCA also optimizes correlated subqueries (CSQs), which are subqueries that reference columns from the outer query. For example, in the following statement, both the inner and outer queries use the `price` column:

```
SELECT * FROM part p1 WHERE price > (SELECT avg(price) FROM part p2 WHERE p2.brand =
p1.brand);
```

GPORCA can generate more optimal execution plans for the following types of subqueries:

- Correlated subqueries in the SELECT list:

```
SELECT *,
(SELECT min(price) FROM part p2 WHERE p1.brand = p2.brand)
AS foo
FROM part p1;
```

- Correlated subqueries in an OR condition:

```
SELECT FROM part p1 WHERE p_size > 40 OR
    p_retailprice >
    (SELECT avg(p_retailprice)
        FROM part p2
        WHERE p2.p_brand = p1.p_brand)
```

- Nested subqueries with skip-level correlations:

```
SELECT * FROM part p1 WHERE p1.p_partkey
IN (SELECT p_partkey FROM part p2 WHERE p2.p_retailprice =
    (SELECT min(p_retailprice)
        FROM part p3
        WHERE p3.p_brand = p1.p_brand)
);
```

- Correlated subqueries with aggregations and inequalities:

```
SELECT * FROM part p1 WHERE p1.p_retailprice =
(SELECT min(p_retailprice) FROM part p2 WHERE p2.p_brand <> p1.p_brand);
```

- Correlated subqueries that must return a single row:

```
SELECT p_partkey,
(SELECT p_retailprice FROM part p2 WHERE p2.p_brand = p1.p_brand )
FROM part p1;
```

Common Table Expression (CTE) optimization

GPORCA efficiently handles queries with `WITH` clauses. A `WITH` clause, also known as a Common Table Expression (CTE), defines a temporary logical table for use within a query. Here is an example of a query containing a CTE:

```
WITH v AS (SELECT a, sum(b) as s FROM T where c < 10 GROUP BY a)
  SELECT * FROM  v AS v1 ,  v AS v2
  WHERE v1.a <> v2.a AND v1.s < v2.s;
```

As part of query optimization, GPORCA supports pushing predicates down into the CTE. For example, in the following query, the equality predicate is pushed down into the CTE:

```
WITH v AS (SELECT a, sum(b) as s FROM T GROUP BY a)
  SELECT *
  FROM v as v1, v as v2, v as v3
  WHERE v1.a < v2.a
    AND v1.s < v3.s
    AND v1.a = 10
    AND v2.a = 20
    AND v3.a = 30;
```

GPORCA supports the following types of CTEs:

- CTEs that define multiple logical tables simultaneously. In the following example, the CTE defines two logical tables:

  ```
  WITH cte1 AS (SELECT a, sum(b) as s FROM T
                  where c < 10 GROUP BY a),
      cte2 AS (SELECT a, s FROM cte1 where s > 1000)
  SELECT *
  FROM cte1 as v1, cte2 as v2, cte2 as v3
  WHERE v1.a < v2.a AND v1.s < v3.s;
  ```

DML operation optimization

GPORCA also enhances DML operations such as `INSERT`, `UPDATE`, and `DELETE`:

- DML operations appear as regular operator nodes in the execution plan.
  - They can appear anywhere in the plan (currently limited to the top-level slice).

    - They can have downstream nodes (consumers).

- `UPDATE` operations are implemented using the Split operator, which supports the following features:

    - Supports updating distribution key columns.

    - Supports updating partition key columns. The following example shows a plan that includes a Split operator:

```
QUERY PLAN
------------------------------------------------------------------
Update  (cost=0.00..5.46 rows=1 width=1)
   ->  Redistribute Motion 2:2  (slice1; segments: 2)
         Hash Key: a
         ->  Result  (cost=0.00..3.23 rows=1 width=48)
               ->  Split  (cost=0.00..2.13 rows=1 width=40)
                     ->  Result  (cost=0.00..1.05 rows=1 width=40)
                           ->  Seq Scan on dmltest
```

Other optimization capabilities

GPORCA also includes the following optimization capabilities:

- Better join ordering choices.
- Support for reordering of joins and aggregate operations.
- Sort order optimization.
- Consideration of data skew estimates during optimization.

What's New in GPORCA Optimizer

This document describes the functional enhancements and behavioral changes of the GPORCA optimizer in release versions of SynxDB Elastic.

v4.0

In v4.0, the GPORCA optimizer added or improved the following features:

- Added support for `FULL JOIN`, executed using `Hash Full Join`. This implementation does not depend on the sorting of join columns and is suitable for cases with large data volumes, high cardinality of join columns, or inconsistent distribution keys.

  Currently, the `Merge Full Join` path is not yet supported, so all `FULL JOIN` queries are executed using `Hash Full Join`.

  Compared to the traditional `Merge Join`, the advantages of `Hash Full Join` include:

  - No need to sort join columns.
  - Can reduce the data transfer overhead of Motion.
  - May have better performance when join columns are unevenly distributed or have high cardinality.

  Example:

  ```
  EXPLAIN SELECT * FROM t1 FULL JOIN t2 ON t1.id = t2.id;
  ```

  May generate a plan like the following:

  ```
  Hash Full Join
  Hash Cond: t1.id = t2.id
  ...
  ```

- GPORCA introduces a query rewrite rule that can push the `JOIN` operation down into each branch of a `UNION ALL`. When this optimization is enabled, the optimizer may decompose the `JOIN` operation into multiple subqueries, leading to the following performance improvements:

  - Supports converting a join on a large table resulting from a `UNION ALL` into multiple joins on smaller tables.
  - The `JOIN` can be pushed down to either the left or right side of the `UNION ALL`, making it applicable to more query structures.

  This optimization is not enabled by default. It can be explicitly enabled with the following

GUC parameter:

```
SET optimizer_enable_push_join_below_union_all = on;
```

The following example shows how the optimizer pushes the `JOIN` down into the `UNION ALL` branches after this optimization is enabled:

```
-- Create test tables
CREATE TABLE dist_small_1(c1 int);
INSERT INTO dist_small_1 SELECT generate_series(1, 1000);
ANALYZE dist_small_1;

CREATE TABLE dist_small_2(c1 int);
INSERT INTO dist_small_2 SELECT generate_series(1, 1000);
ANALYZE dist_small_2;

CREATE TABLE inner_1(cc int);
INSERT INTO inner_1 VALUES(1);
ANALYZE inner_1;

-- Create a view
CREATE VIEW dist_view_small AS
SELECT c1 FROM dist_small_1
UNION ALL
SELECT c1 FROM dist_small_2;

-- Enable optimization and execute the query
SET optimizer_enable_push_join_below_union_all = on;
EXPLAIN ANALYZE
SELECT c1 FROM dist_view_small JOIN inner_1 ON c1 < cc;
```

This optimization is applicable to the following types of query structures:

- Multiple large tables are aggregated via `UNION ALL` and then joined with a small table.
- One side of the join in the query structure is a view or a `UNION ALL` subquery.

Note

- Currently, this optimization does not support `FULL JOIN` and Common Table

Expressions (CTEs).

- Structures like 'JOIN of UNION ALL' and 'UNION ALL of JOIN' are also not yet supported.

- By default, GPORCA sets a higher cost for the broadcast path (Broadcast Motion) based on the `optimizer_penalize_broadcast_threshold` GUC parameter to prevent selecting overly expensive plans for large data volumes.

  For `NOT IN` type queries (i.e., Left Anti Semi Join, LASJ), the broadcast path is no longer penalized. This optimization prevents the optimizer from, in some cases, concentrating large tables on the coordinator node for execution, which can cause severe performance issues or even out-of-memory (OOM) errors.

  Allowing the use of the broadcast path preserves parallel execution and significantly improves the execution efficiency of `NOT IN` queries with large data volumes.

  Feature description:

  - Only affects `NOT IN` queries (LASJ).
  - Ignores the setting of `optimizer_penalize_broadcast_threshold`.
  - The penalty policy is still retained for other types of joins (such as `IN` or `EXISTS`).

  Example:

```
SELECT * FROM foo WHERE a NOT IN (SELECT a FROM bar);
```

  Example query plan:

```
Gather Motion 2:1
-> Hash Left Anti Semi (Not-In) Join
    -> Seq Scan on foo
    -> Broadcast Motion
            -> Seq Scan on bar
```

Optimization for multi-level outer self-joins

GPORCA can identify certain specific patterns of multi-level outer joins and skip unnecessary Redistribute Motion to improve execution efficiency:

- The query structure contains multiple `LEFT OUTER JOIN`s or `RIGHT OUTER JOIN`s.
- All tables involved in the join are aliases of the same base table.
- The join condition is a symmetric condition (e.g., `t1.a = t2.a`).
- All tables use the same distribution key, and the data distribution meets locality requirements.

Example:

```
CREATE TABLE o1 (a1 int, b1 int);

EXPLAIN (COSTS OFF)
SELECT * FROM (SELECT DISTINCT a1 FROM o1) t1
             LEFT OUTER JOIN o1 t2 ON t1.a1 = t2.a1
             LEFT OUTER JOIN o1 t3 ON t2.a1 = t3.a1;
```

GPORCA can recognize this multi-level self-join structure and avoid redundant data redistribution, thereby improving overall query performance.

5.1.3 Update Statistics

Accurate statistics are crucial for good query performance. Using the `ANALYZE` statement to update statistics allows the query optimizer to generate optimal query plans. When SynxDB Elastic analyzes a table, the relevant data information is stored in the system catalog tables. If this stored information becomes outdated, the query optimizer may generate inefficient query plans.

Check whether statistics are updated

To check whether a table's statistics are up-to-date, you can use the `pg_stat_all_tables` system view. The `last_analyze` column in this view shows the last time the table was manually analyzed, while the `last_autoanalyze` column shows the last time it was auto-analyzed. The timestamps in both columns are updated when an `ANALYZE` statement is executed.

For example, to check whether the statistics for the `test_analyze` table are updated, you can execute the following query:

```
SELECT schemaname, relname, last_analyze, last_autoanalyze
FROM pg_stat_all_tables
WHERE relname = 'test_analyze';
```

Selectively generate statistics

Executing `ANALYZE` without any arguments updates the statistics for all tables in the database. This is a very time-consuming process and is not recommended. It is advisable to selectively run `ANALYZE` on tables when data has changed, or to use the `analyzedb` utility.

Running `ANALYZE` on a large table can take a long time. If you cannot run `ANALYZE` on all columns of a large table, you can generate statistics for specific columns only using `ANALYZE table(column, ...)`. Make sure to include columns that are used in joins, `WHERE` clauses, `SORT` clauses, `GROUP BY` clauses, or `HAVING` clauses.

For partitioned tables, you can run `ANALYZE` only on the partitions that have changed, for example, when a new partition is added. Note that for a partitioned table, you can run `ANALYZE` on the root partition table or on a leaf partition (the file that actually stores the data and statistics). In SynxDB Elastic, running `ANALYZE` on a single partition of a partitioned table also updates the statistics of the root table, which means collecting statistics on one partition can affect the optimizer statistics for the entire partitioned table. You can use the `pg_partition_tree()` function to find the names of the leaf partitions.

```
SELECT * FROM pg_partition_tree( 'parent_table' );
```

Improve statistics quality

There is a trade-off between the time it takes to generate statistics and the quality or accuracy of those statistics. You need to find a balance.

To analyze large tables in a reasonable amount of time, `ANALYZE` takes a random sample of the table contents rather than checking every row. To increase the number of sample values for all columns on a table, you can adjust the `default_statistics_target` configuration parameter. The target value for this parameter ranges from `1` to `10000`, with a default value of `100`.

By default, the `default_statistics_target` parameter applies to all columns and specifies how many values are stored in the list of most common values. A larger target value can improve the quality of the query planner's estimates, especially for columns with irregular data patterns.

You can set `default_statistics_target` at the session level using the `SET default_statistics_target` statement. To set the default value for this parameter, you need to set it in the `postgresql.conf` configuration file and then reload the file.

When to run ANALYZE

Run `ANALYZE` in the following situations:

- After loading data
- After performing `INSERT`, `UPDATE`, and `DELETE` operations that significantly change the underlying data

`ANALYZE` only requires a read lock on the table, so it can be run in parallel with other database operations. However, for performance reasons, it is not recommended to run `ANALYZE` at the same time as loading, `INSERT`, `UPDATE`, and `DELETE` operations.

> **Note**
>
> SynxDB Elastic has optimized the behavior of running `ANALYZE` on partitioned tables. When statistics are explicitly collected on a leaf partition (e.g., `ANALYZE sales_1_prt_p2023`), the system no longer automatically updates the statistics of the root table or other partitions. Only when `ANALYZE` is explicitly run on the root table (e.g., `ANALYZE sales`) will the statistics for the entire table, including all sub-partitions, be updated.
>
> This behavioral improvement provides greater control over statistics management and avoids unnecessary statistical refreshes. In practice, it is recommended to selectively analyze specific partitions or the entire table based on data changes.

Configure automatic statistics collection

The `gp_autostats_mode` configuration parameter, in conjunction with `gp_autostats_on_change_threshold`, determines when to trigger an automatic analysis operation. When automatic statistics collection is triggered, the optimizer adds an `ANALYZE` step to the query.

By default, the value of `gp_autostats_mode` is `none`. If this parameter is set to `on_no_stats`, statistics collection will be triggered for `CREATE TABLE AS SELECT`, `INSERT`, or `COPY` operations on tables that have no existing statistics, when performed by the table owner.

When `gp_autostats_mode` is set to `on_change`, statistics are collected only when the number of affected rows exceeds the threshold set by `gp_autostats_on_change_threshold`. The default value for this threshold is `2147483647`. Automatic statistics collection is triggered when the number of affected rows in `CREATE TABLE AS SELECT`, `UPDATE`, `DELETE`, `INSERT`, and `COPY` operations, performed by the table owner, exceeds this threshold.

In addition, if the `gp_autostats_allow_nonowner` server configuration parameter is set to `true`, SynxDB Elastic will perform automatic statistics collection on a table in the following case:

- When `gp_autostats_mode` is set to `on_no_stats`, and a non-owner user is the first to perform an `INSERT` or `COPY` operation on the table.

Setting `gp_autostats_mode` to `none` disables automatic statistics collection.

For partitioned tables, inserting data into the top-level parent table does not trigger automatic statistics collection. However, if data is inserted directly into a leaf table of the partitioned table (where the data is actually stored), automatic statistics collection will be triggered.

5.1.4 Query Hints

SynxDB Elastic uses two query optimizers: the Postgres-based optimizer and GPORCA. Each optimizer is tailored for specific types of workloads:

- The Postgres-based optimizer: Suitable for transactional workloads.
- GPORCA: Suitable for analytical and hybrid transactional-analytical workloads.

When processing a query, the optimizer explores a vast search space of equivalent execution plans. It uses table statistics and cardinality estimation models to predict the number of rows processed by each operation. The optimizer then assigns a cost to each plan using a cost model and selects the one with the lowest cost as the final execution plan.

Query hints are directives given by the user to the optimizer to influence the query's execution strategy. Hints allow users to override the optimizer's default behavior to address issues like inaccurate row count estimates, suboptimal scan methods, inappropriate join type selection, or inefficient join orders. This article introduces the different types of query hints and their use cases.

Quick example

```
CREATE TABLE foo(a int);
INSERT INTO foo SELECT i FROM generate_series(1, 100000)i;

LOAD 'pg_hint_plan';
SHOW pg_hint_plan.enable_hint;
pg_hint_plan.enable_hint
--------------------------
on
(1 row)

EXPLAIN SELECT count(*) FROM foo WHERE a > 6;
                                    QUERY PLAN
-----------------------------------------------------------------------------------------
Finalize Aggregate  (cost=537.05..537.06 rows=1 width=8)
   ->  Gather Motion 3:1  (slice1; segments: 3)  (cost=536.99..537.04 rows=3 width=8)
         ->  Partial Aggregate  (cost=536.99..537.00 rows=1 width=8)
               ->  Seq Scan on foo  (cost=0.00..453.67 rows=33330 width=0)
                     Filter: (a > 6)
Optimizer: Postgres-based planner
(6 rows)
```

You can also specify multiple hints at once, for example, to control the scan method and row count estimation:

```
SELECT * FROM t1 JOIN t2 ON t1.a = t2.a WHERE t1.a < 100;
```

Cardinality hints

When the optimizer's row count estimation for a join operation is inaccurate, it might choose a less efficient plan, such as using Broadcast Motion instead of Redistribute Motion, or incorrectly favoring a Merge Join over a Hash Join. Cardinality hints can be used to adjust the row count estimate for a specific operation, which is particularly useful when statistics are missing or outdated.

Example:

```
/*+ Rows(t1 t2 t3 #42) */ SELECT * FROM t1, t2, t3;  -- Sets the estimated rows to 42
/*+ Rows(t1 t2 t3 +42) */ SELECT * FROM t1, t2, t3; -- Adds 42 to the original
```

(continues on next page)

(continued from previous page)

```
estimate
/*+ Rows(t1 t2 t3 -42) */ SELECT * FROM t1, t2, t3; -- Subtracts 42 from the original
estimate
/*+ Rows(t1 t2 t3 *42) */ SELECT * FROM t1, t2, t3; -- Multiplies the original
estimate by 42
```

Cardinality hints currently only take effect in the ORCA optimizer. The Postgres optimizer does not recognize these hints.

Join type hints

When using a Hash Join, some intermediate results might be written to disk, affecting performance. If a user knows that a specific query is better suited for a Nested Loop Join, they can specify the join method and the order of the inner and outer tables using hints.

Example:

```
/*+ HashJoin(t1 t2) */ SELECT * FROM t1, t2;
/*+ NestLoop(t1 t2) */ SELECT * FROM t1, t2;
/*+ MergeJoin(t1 t2) */ SELECT * FROM t1 FULL JOIN t2 ON t1.a = t2.a;
```

Join order hints

When the optimizer chooses an inefficient join order due to insufficient statistics or estimation biases, you can use the `Leading(...)` hint to specify the join order between tables.

Example:

```
/*+ Leading(t1 t2 t3) */ SELECT * FROM t1, t2, t3;
/*+ Leading(t1 (t3 t2)) */ SELECT * FROM t1, t2, t3;
```

In queries involving `LEFT OUTER JOIN` or `RIGHT OUTER JOIN`, you can also use `Leading(...)` to specify the join order. Note the following restrictions when using it:

- The hint order must be consistent with the join structure in the original SQL. For example:

  ```
  SELECT * FROM t1 LEFT JOIN t2 ON t1.a = t2.a;
  ```

  Using `/*+ Leading((t1 t2)) */` preserves the left join, while `/*+ Leading((t2 t1)) */` converts it to a right join (semantically equivalent, but a different plan).

- For multi-level nested outer joins, the hint must be specified in the semantic nesting order.
- Adjusting the join direction is not supported for non-equi-join conditions (e.g., `t1.a > t2.a`), as this would alter the query's semantics.

  Example: The following hint instructs the optimizer to prioritize joining `t3` with the `t2-t1` combination:

```
/*+ Leading((t3 (t2 t1))) */
SELECT * FROM t1 LEFT JOIN t2 ON t1.a = t2.a LEFT JOIN t3 ON t2.b = t3.b;
```

Scope and limitations of hints

- The query hints feature depends on the `pg_hint_plan` extension module, which must be explicitly loaded.
- Hints for controlling data redistribution strategies are not currently supported.

Best practices for query hints

It is recommended to follow these practices when using hints:

- Focus on solving specific problems: such as inaccurate cardinality estimates, poor scan methods, or suboptimal join types or orders.
- Test thoroughly before deploying to production: Ensure the hint genuinely improves query performance and reduces resource consumption.
- Use as a temporary measure: Hints should be used for short-term optimization and reviewed periodically as data changes.
- Avoid conflicts with GUCs: If a GUC setting conflicts with a hint, the hint will be ignored. Ensure global configurations are consistent with your hints.

5.1.5 Automatically Use Materialized Views for Query Optimization

SynxDB Elastic supports automatically using *materialized views* during the query planning phase to compute part or all of a query (a feature known as AQUMV). This is particularly useful for queries on large tables, significantly improving query processing time. AQUMV utilizes incremental materialized views (IMVs) because they generally stay up-to-date when the underlying tables have write operations.

Scenarios

- **Large-scale aggregate queries**: For queries that need to aggregate millions of records, AQUMV can significantly reduce query time.
- **Frequently updated large tables**: In environments with frequent data updates, using IMVs ensures the timeliness and accuracy of query results.
- **Complex calculation scenarios**: For queries involving complex calculations (like the square root and absolute value calculations in the example below), AQUMV can speed up queries by pre-calculating these values in a materialized view.
- **High-frequency data write scenarios**: When the base table has very frequent data writes, synchronously updating the materialized view might greatly impact write performance. In this case, you can use asynchronous incremental materialized views to run refresh tasks in the background, reducing the impact on foreground business operations.
- **Aggregate queries on large partitioned tables**: Partitioned tables are often used to manage large amounts of data. Building materialized views on large partitioned tables allows complex aggregate queries or join operations across multiple partitions to be pre-computed. The AQUMV feature can automatically leverage these materialized views, improving query analysis performance on large partitioned tables and avoiding the significant overhead of full table scans.

Example

To enable the AQUMV feature, you first need to create a materialized view and set the system parameter `enable_answer_query_using_materialized_views` to `ON`. The following is a comparison of executing the same complex query with and without AQUMV.

1. Disable the GPORCA optimizer and use the Postgres-based planner.

```
SET optimizer TO off;
```

2. Create the table `aqumv_t1`.

```
CREATE TABLE aqumv_t1(c1 INT, c2 INT, c3 INT);
```

3. Insert data into the table and collect statistics on it.

```
INSERT INTO aqumv_t1 SELECT i, i+1, i+2 FROM generate_series(1, 100000000) i;
ANALYZE aqumv_t1;
```

4. Execute the query without AQUMV enabled. It takes 7384.329 ms.

```
SELECT SQRT(ABS(ABS(c2) - c1 - 1) + ABS(c2)) FROM aqumv_t1 WHERE c1 > 30 AND c1 <
40 AND SQRT(ABS(c2)) > 5.8;

    sqrt
-------------------66.0827625302982196.2449979983983985.9160797830996166.
1644140029689766.3245553203367595.830951894845301
(7 rows)

Time: 7384.329 ms (00:07.384)
```

   The query plan is as follows, showing that the optimizer performs a scan on the table (`Seq Scan on aqumv_t1`).

```
EXPLAIN(COSTS OFF) SELECT SQRT(ABS(ABS(c2) - c1 - 1) + ABS(c2)) FROM aqumv_t1
WHERE c1 > 30 AND c1 < 40 AND SQRT(ABS(c2)) > 5.8;

                                                QUERY PLAN

-----------------------------------------------------------------------------------
---------------------------
Gather Motion 3:1  (slice1; segments: 3)
->  Seq Scan on aqumv_t1
        Filter: ((c1 > 30) AND (c1 < 40) AND (sqrt((abs(c2))::double precision) >
'5.8'::double pre
cision))
Optimizer: Postgres query optimizer
(4 rows)
```

5. Create a materialized view `mvt1` based on `aqumv_t1` and collect statistics on the view.

```
CREATE INCREMENTAL MATERIALIZED VIEW mvt1 AS SELECT c1 AS mc1, c2 AS mc2, ABS(c2)
AS mc3, ABS(ABS(c2) - c1 - 1) AS mc4
FROM aqumv_t1 WHERE c1 > 30 AND c1 < 40;

ANALYZE mvt1;
```

In the example above, the materialized view `mvt1` is updated synchronously. If data is inserted into the `aqumv_t1` table very frequently, you can change it to asynchronous refresh mode to improve write performance.

Just add the `REFRESH DEFERRED` and `SCHEDULE` clauses during creation:

```
-- Changes the materialized view to asynchronous refresh with a 10-second
interval.
CREATE INCREMENTAL MATERIALIZED VIEW mvt1
REFRESH DEFERRED SCHEDULE '10 seconds'
AS SELECT c1 AS mc1, c2 AS mc2, ABS(c2) AS mc3, ABS(ABS(c2) - c1 - 1) AS mc4
FROM aqumv_t1 WHERE c1 > 30 AND c1 < 40;
```

This way, the data in the materialized view will be periodically updated by a background task, and the query acceleration feature (AQUMV) will still automatically use this up-to-date materialized view.

6. Enable the AQUMV-related configuration parameter:

```
SET enable_answer_query_using_materialized_views = ON;
```

7. Now that AQUMV is enabled, execute the same query again. It takes 45.701 ms.

```
SELECT SQRT(ABS(ABS(c2) - c1 - 1) + ABS(c2)) FROM aqumv_t1 WHERE c1 > 30 AND c1 <
40 AND SQRT(ABS(c2)) > 5.8;

    sqrt
--------------------66.0827625302982196.2449979983983985.8309518948453015.
9160797830996166.1644140029689766.324555320336759
(7 rows)

Time: 45.701 ms
```

The query plan is as follows, showing that the optimizer scans the materialized view `mvt1` (`Seq Scan on public.mvt1`) instead of the table `aqumv_t1`.

```
explain(verbose, costs off)select sqrt(abs(abs(c2) - c1 - 1) + abs(c2)) from
aqumv_t1 where c1 > 30 and c1 < 40 and sqrt(abs(c2)) > 5.8;

                                QUERY PLAN
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------------------------------------
Gather Motion 3:1  (slice1; segments: 3)
Output: (sqrt(((mc4 + mc3))::double precision))
->  Seq Scan on public.mvt1
        Output: sqrt(((mc4 + mc3))::double precision)
        Filter: (sqrt((mvt1.mc3)::double precision) > '5.8'::double precision)
Settings: enable_answer_query_using_materialized_views = 'on', optimizer = 'off'
Optimizer: Postgres query optimizer
(7 rows)
```

In the example above, the query took 7384.329 ms without using a materialized view. After enabling AQUMV, the same query using the materialized view took only 45.701 ms. This demonstrates that the materialized view greatly improves performance by pre-calculating and storing the relevant computation results, containing only the rows that satisfy the specific condition (`c1 > 30 and c1 < 40`).

Therefore, the table query `select sqrt(abs(abs(c2) - c1 - 1) + abs(c2)) from aqumv_t1 where c1 > 30 and c1 < 40 and sqrt(abs(c2)) > 5.8;` is actually equivalent to the query on the materialized view `select sqrt(mc4 + mc3) from mvt1 where sqrt(mc3) > 5.8;`.

This way, when executing the same query, data can be fetched directly from the materialized view instead of the original table. This allows AQUMV to greatly improve query performance, especially when dealing with large data volumes and complex calculations.

How it works

AQUMV achieves query optimization by performing an equivalent transformation of the query tree.

SynxDB Elastic automatically uses a materialized view for a table query only if the following conditions are met:

- The materialized view must contain all the rows required by the query expression.
- If the materialized view contains more rows than the query, additional filtering conditions might need to be added.
- All output expressions must be computable from the view’ s output.

- The output expressions can fully or partially match the target list of the materialized view.

When there are multiple valid materialized view candidates, or when the cost of querying from the materialized view is higher than querying directly from the original table, the planner can decide the best choice based on cost estimation.

Limitations

- Only `SELECT` queries on a single relation are supported, applicable to both the materialized view query and the original query.
- Currently, the following features are not supported: aggregations (AGG), subqueries, sorting in the original query (ORDER BY), joins (JOIN), sublinks (SUBLINK), grouping (GROUP BY), window functions, Common Table Expressions (CTE), deduplication (DISTINCT ON), `REFRESH MATERIALIZED VIEW`, and `CREATE AS` statements.
- Incremental materialized views are currently optimized primarily for data append (`INSERT`) scenarios. If `DELETE`, `UPDATE`, or `TRUNCATE` operations occur on its base table, the materialized view will perform a costly full refresh instead of an incremental update.

5.1.6 Push Down Aggregation Operations

Aggregation pushdown is an optimization technique that moves aggregation operations closer to the data source. SynxDB Elastic supports pushing down aggregations by performing the aggregation operator before the join operator.

In appropriate scenarios, aggregation pushdown can significantly reduce the size of the input set for the join or aggregation operator, thereby improving the operator's execution performance.

Note

- In the native PostgreSQL optimizer logic, aggregation operations in a query are always performed after all join operations have been completed (excluding subqueries). Therefore, SynxDB Elastic introduces the aggregation pushdown feature, allowing it to choose to perform aggregation operations earlier in suitable scenarios.
- To determine whether the execution plan chosen by the optimizer has applied the aggregation pushdown optimization, you can observe the positional relationship between Aggregation and Join in the execution plan tree. If an execution plan first performs a Partial Aggregation, then a Join, and finally a Final Aggregation, it means the optimizer

has applied aggregation pushdown.

Usage example

Before using this optimization, you need to manually enable the GUC parameter `gp_enable_agg_pushdown`.

Additionally, you need to manually set `optimizer=off` to disable the GPORCA optimizer, as this optimization currently only works in the PostgreSQL optimizer.

Here is an example of using aggregation pushdown optimization.

```
-- Creates two tables, t1 and t2.
CREATE TABLE t1(id INT, val1 INT);
CREATE TABLE t2(id INT, val2 INT);

SET OPTIMIZER=OFF; -- Disable the GPORCA optimizer
SET gp_enable_agg_pushdown=ON; -- Enable the GUC parameter

-- Executes a query with aggregation and join operations.
EXPLAIN (COSTS OFF) SELECT id, SUM(val1) FROM t1 NATURAL JOIN t2 GROUP BY id;
                     QUERY PLAN
-----------------------------------------------------
 Gather Motion 3:1  (slice1; segments: 3)
   ->  Finalize GroupAggregate
         Group Key: t1.id
         ->  Sort
               Sort Key: t1.id
               ->  Hash Join
                     Hash Cond: (t2.id = t1.id)
                     ->  Seq Scan on t2
                     ->  Hash
                           ->  Partial HashAggregate
                                 Group Key: t1.id
                                 ->  Seq Scan on t1
Optimizer: Postgres query optimizer
(13 rows)
```

From the execution plan result in the example above, you can see that before performing the HashJoin operation, SynxDB Elastic first performs an aggregation on the `t1` table based on the `id`

column. This aggregation does not compromise the correctness of the statement result and is likely to reduce the amount of data entering the HashJoin, thus improving the statement's execution efficiency.

Applicable scenarios

Using aggregation pushdown in the following scenarios is expected to yield significant query performance improvements.

Scenario 1

Scenario description: Each record in one table corresponds to multiple records in another table, and the two tables need to be joined for a grouped aggregation.

For example, you need to join an `order_tbl` with an `order_line_tbl` and calculate the total amount for each order by summing the prices of its corresponding order items, i.e., `SUM(price)`:

```
SELECT o.order_id, SUM(price)
  FROM order_tbl o, order_line_tbl ol
  WHERE o.order_id = ol.order_id
  GROUP BY o.order_id;
```

- **Execution in the native PostgreSQL optimizer**: The native PostgreSQL optimizer can only join the two tables first and then perform the aggregation. Because every order item in `order_line_tbl` must have a corresponding order in `order_tbl`, the Join operator will not filter out any data.
- **With aggregation pushdown**: Assuming each order contains an average of 10 order items, the data volume is expected to decrease by a factor of 10 after the Aggregation operator. With aggregation pushdown enabled, the database will first aggregate the data in `order_line_tbl` by `order_id`. This reduces the amount of data passed to the Join operator by a factor of 10, significantly lowering the cost of the Join. The corresponding execution plan is as follows:

```
EXPLAIN SELECT o.order_id, SUM(price)
FROM order_tbl o, order_line_tbl ol
WHERE o.order_id = ol.order_id
GROUP BY o.order_id;
                                        QUERY PLAN
-------------------------------------------------------------------------------
-------------
```

(continues on next page)

(continued from previous page)

```
Gather Motion 3:1  (slice1; segments: 3)  (cost=712.89..879.56 rows=10000
width=12)
->  Finalize HashAggregate  (cost=712.89..746.23 rows=3333 width=12)
        Group Key: o.order_id
        ->  Hash Join  (cost=617.00..696.23 rows=3333 width=12)
            Hash Cond: (ol.order_id = o.order_id)
            ->  Partial HashAggregate  (cost=538.00..571.38 rows=3338 width=12)
                    Group Key: ol.order_id
                    ->  Seq Scan on order_line_tbl ol  (cost=0.00..371.33
rows=33333 width=8)
            ->  Hash  (cost=37.33..37.33 rows=3333 width=4)
                    ->  Seq Scan on order_tbl o  (cost=0.00..37.33 rows=3333
width=4)
Optimizer: Postgres query optimizer
```

A similar scenario is joining a `project` table with an `experiment` table to calculate the total experiment cost for each project over the past year. The reference SQL statement is as follows:

```
SELECT proj_name, sum(cost)
    FROM experiment e, project p
    WHERE e.e_pid = p.p_pid AND e.start_time > now() - interval '1 year'
    GROUP BY proj_name;
```

For this query, with aggregation pushdown enabled, SynxDB Elastic will pre-aggregate the `experiment` table by the `e_pid` column, grouping information for the same project first.

However, if this query also involves significant filtering on the `project` table, the join selectivity might become too high, leading to inefficient execution. Therefore, aggregation pushdown is not currently suitable for this situation.

Scenario 2

Scenario description: The Join operator in the query significantly expands the result set, which ultimately needs to be grouped for calculation.

For example, joining a `person_1` table with a `person_2` table to find out how many different pairs can be formed for each common name between the two tables:

```
SELECT p1.name, COUNT(p1.name) FROM person_1 p1, person_2 p2 WHERE p1.name = p2.name
GROUP BY p1.name;
```

In this example, if a `name` appears X times in the `p1` table and Y times in the `p2` table, that `name` will appear X*Y times in the final result. If a large amount of data fits this pattern, the result set after the join could be very large.

In the example above, if the aggregation is pushed down to either the `p1` or `p2` side, each `name` will appear at most once after aggregation on that side. This effectively reduces the cost of the Join operator and the size of the input set for the subsequent Aggregation operator. The corresponding execution plan is as follows:

```
EXPLAIN SELECT p1.name, COUNT(p1.name) FROM person_1 p1, person_2 p2 WHERE p1.name =
p2.name GROUP BY p1.name;
                                        QUERY PLAN
-------------------------------------------------------------------------------------
---
 Gather Motion 3:1  (slice1; segments: 3)  (cost=1758.62..1925.23 rows=9997 width=12)
   ->  Finalize HashAggregate  (cost=1758.62..1791.94 rows=3332 width=12)
         Group Key: p1.name
         ->  Hash Join  (cost=762.93..1592.17 rows=33290 width=12)
               Hash Cond: (p2.name = p1.name)
               ->  Seq Scan on p2  (cost=0.00..371.33 rows=33333 width=4)
               ->  Hash  (cost=637.97..637.97 rows=9997 width=12)
                     ->  Partial HashAggregate  (cost=538.00..637.97 rows=9997
width=12)
                           Group Key: p1.name
                           ->  Seq Scan on p1  (cost=0.00..371.33 rows=33333 width=4)
 Optimizer: Postgres query optimizer
(11 rows)
```

Scenarios where it is not recommended

Aggregation pushdown is unlikely to provide performance benefits in the following scenarios and is not recommended.

Not recommended scenario 1

Scenario description: Scenarios where aggregation does not significantly change the data volume.

Contrary to the applicable scenarios above, if performing the aggregation early does not change the data volume and cannot reduce the input set size for subsequent calculations, the Join operator should be executed first to avoid unnecessary overhead.

Not recommended scenario 2

Scenario description: If the join key is different from the grouping key, aggregation pushdown will cause the grouping key to change after being pushed down. In this case, the aggregation after rewriting the grouping key cannot reduce the data volume, leading to poor pushdown effectiveness:

```
SELECT t1.value, COUNT(*) FROM t1, t2 WHERE t1.key = t2.key GROUP BY t1.value;
```

For the query example above, directly pushing down the aggregation to the `t1` side would lead to incorrect results, similar to the situation in Limitation 1. To ensure the correctness of the calculation, the actual grouping key for the pushed-down aggregation would be equivalent to `GROUP BY t1.key, t1.value`.

In this case, if the `key` and `value` in the `t1` table are completely unrelated, each group might contain only a single tuple, so this aggregation pushdown will not have any positive effect. However, if `key` and `value` are strongly correlated, or if the same `key` always corresponds to the same `value`, the grouping effectiveness will not be affected.

In the example above, grouping by `t1.value` was originally effective. But after aggregation pushdown, the grouping key becomes `t1.key, t1.value`. If `key` and `value` have a weak correlation, this aggregation will not produce a significant effect.

Usage limitations

This section describes some limitations of the aggregation pushdown feature, including cases where this optimization is logically inapplicable and cases that are not yet supported by the engineering implementation.

Limitation 1

Limitation description: Aggregation pushdown cannot be applied when filtering is performed on columns other than the `GROUP BY` columns during the join and subsequent calculations. Consider the following SQL query:

```
SELECT id, SUM(val) FROM t1, t2 WHERE t1.id = t2.id AND t1.val > t2.val GROUP BY id;
```

In the example above, assume there are two tuples A and B from the `t1` table with `id = 100`, and a tuple C from the `t2` table also with `id = 100`.

During the join of AB and C, even though A and B have the same `id`, it is not guaranteed that they will both pass or fail the `AB.val > C.val` filter condition simultaneously. In this situation, pre-aggregating the `val` based on `id` would inevitably sum the `val` of A and B together. However, since they do not necessarily pass or fail the filter condition at the same time, this would lead to incorrect results.

In contrast, the following similar query example can apply aggregation pushdown:

```
SELECT id, SUM(val) FROM t1, t2 WHERE t1.id = t2.id AND t1.id < t2.id_thre GROUP BY
id;
```

This example also considers the same three tuples A, B, and C as in the previous example. Because the additional filter condition only uses the `id` column from `t1`, when the two tuples A and B with the same `id` are joined with tuple C, they will either both pass the filter or both fail. Therefore, the `val` of tuples A and B can be summed up in advance through an aggregation operation.

Limitation 2

Limitation description: Pushing down aggregations to both sides of a Join at the same time is not supported. Consider the following SQL query:

```
SELECT id, SUM(val) FROM t1, t2 WHERE t1.id = t2.id GROUP BY id;
```

We can actually rewrite the statement to get an equivalent one:

```
SELECT id, sum1 * cnt2 FROM
    (SELECT id, SUM(val) FROM t1 GROUP BY id) AT1(id, sum1),
    (SELECT id, COUNT(*) FROM t2 GROUP BY id) AT2(id, cnt2)
WHERE AT1.id = AT2.id GROUP BY id;
```

In this example, the aggregation operation is pushed down to both sides of the Join. For all tuples in `t1` with `id = 100`, SynxDB Elastic pre-aggregates their `val` to get the corresponding `sum1`.

During the actual join process, for each tuple in `t2` with `id = 100`, it will be joined with the tuple containing `sum1` to produce a corresponding result tuple. This means that for every `id = 100` in `t2`, `sum1` will appear once in the final summation. Therefore, SynxDB Elastic can pre-aggregate `t2` to calculate that there are a total of `cnt2` tuples with `id = 100`, and finally calculate the result using `sum1 * cnt2`.

Because this scenario involves relatively complex statement and expression rewriting, it is not currently supported in the product.

5.1.7 Execute Queries in Parallel

This document describes the use cases, methods, limitations, and common issues for the parallel query feature in SynxDB Elastic. Parallel query aims to improve query performance by utilizing multiple CPU cores to process a single query.

Use cases

You can deploy a small number of segments on a single physical machine and use dynamic parallelism adjustment as an alternative to deploying a large number of segments, thereby improving performance.

Enabling operator parallelism provides a performance advantage when the host CPU and disk loads are not high.

How to use

SynxDB Elastic supports parallel query on heap tables.

Parallel query on heap tables

1. Before enabling the parallel query feature, you need to disable the GPORCA optimizer.

```
SET enable_parallel = ON;
SET optimizer = OFF;
```

2. Set the maximum degree of parallelism.

```
-- This setting should take into account the number of CPU cores and segments.
SET max_parallel_workers_per_gather = 4;
```

Query example:

```
CREATE TABLE t1 (c1 int,c2 int, c3 int, c4 box);
INSERT INTO t1 SELECT x, 2*x, 3*x, box('6,6,6,6') FROM generate_series(1,1000000) AS
x;
SELECT count(*) from t1;
```

Parameter description

Parameter name	Description	Default value	Required to set	Example
`enable_parallel`	Enables or disables the parallel feature.	`OFF`	Yes	`SET enable_parallel = ON;`
`optimizer`	Enables or disables the GPORCA optimizer.	`ON`	Yes	`SET optimizer = OFF;`

Frequently asked questions

- Currently, parallel execution is supported for queries containing the following operators. SynxDB Elastic does not yet support queries with other operators.

```
sequence scan
bitmap heap scan
append
hash join
nestloop join
merge join
```

- Parallel query does not improve query performance in all situations. An excessively high degree of parallelism can cause excessive load, leading to a decrease in performance.
- Enabling parallelism means a multiplied memory overhead, which may lead to "out of memory" errors.

5.1.8 Auto-Clustering Tables

The `CLUSTER` statement physically reorders data in a table based on a specified index. This sorted data can greatly reduce the scan range during queries, thereby improving query performance. However, physical reordering is a one-time operation. As the table data is updated, its physical order gradually degrades, requiring a database administrator to manually execute the `CLUSTER` command to re-optimize it.

To reduce manual maintenance, SynxDB Elastic provides an auto-clustering feature that automatically reorders tables physically. This feature is enabled by default and uses a background service to periodically and automatically perform incremental `CLUSTER` operations on tables that require physical reordering, thus continuously maintaining an efficient physical data layout.

User value

You will mainly notice its presence and value in the following ways:

- **Improved query performance**: If a query's filter conditions can utilize the cluster key, the amount of data scanned is greatly reduced, leading to a notable increase in query speed.
- **Non-blocking write operations**: The background reordering task runs concurrently with foreground write operations like `INSERT` and `COPY`, ensuring that they do not block each other and maintaining business continuity.
- **Background resource consumption**: Auto-clustering consumes a certain amount of CPU, memory, and I/O. You can observe periodic resource usage peaks in the system through monitoring tools. The intensity and frequency of these peaks can be adjusted using global parameters.
- **Detailed logging**: The database log records details of each reordering operation (such as the table operated on, time taken, and amount of data processed), providing a basis for performance evaluation and troubleshooting.

How to use

In most cases, you only need to define a cluster key for your table. You can also adjust or disable the feature using GUC parameters.

Step 1: Define a cluster key for the table

You need to explicitly specify the columns to be used for physical reordering (the cluster key) when creating (`CREATE TABLE`) or altering (`ALTER TABLE`) a table. The auto-clustering service only affects tables with an explicitly defined cluster key. **If a table does not have a cluster key defined, the service will ignore the table.**

How to choose a cluster key

It is recommended to choose the most frequently used filter columns as the cluster key, based on your business query patterns. For example, columns that often appear in `WHERE` clauses or `JOIN` conditions, such as date columns or region ID columns.

```
-- Example: Creates a table that is physically reordered by columns c1 and c2.
-- This table will be managed by the auto-clustering service.
CREATE TABLE my_table (
    c1 INT,
    c2 INT,
    c3 VARCHAR(50)
) WITH (cluster_columns = 'c1, c2');
```

Step 2: Adjust global configuration parameters

Database administrators can adjust the overall behavior of auto-clustering by setting the following GUC parameters to balance performance optimization effects with system resource consumption.

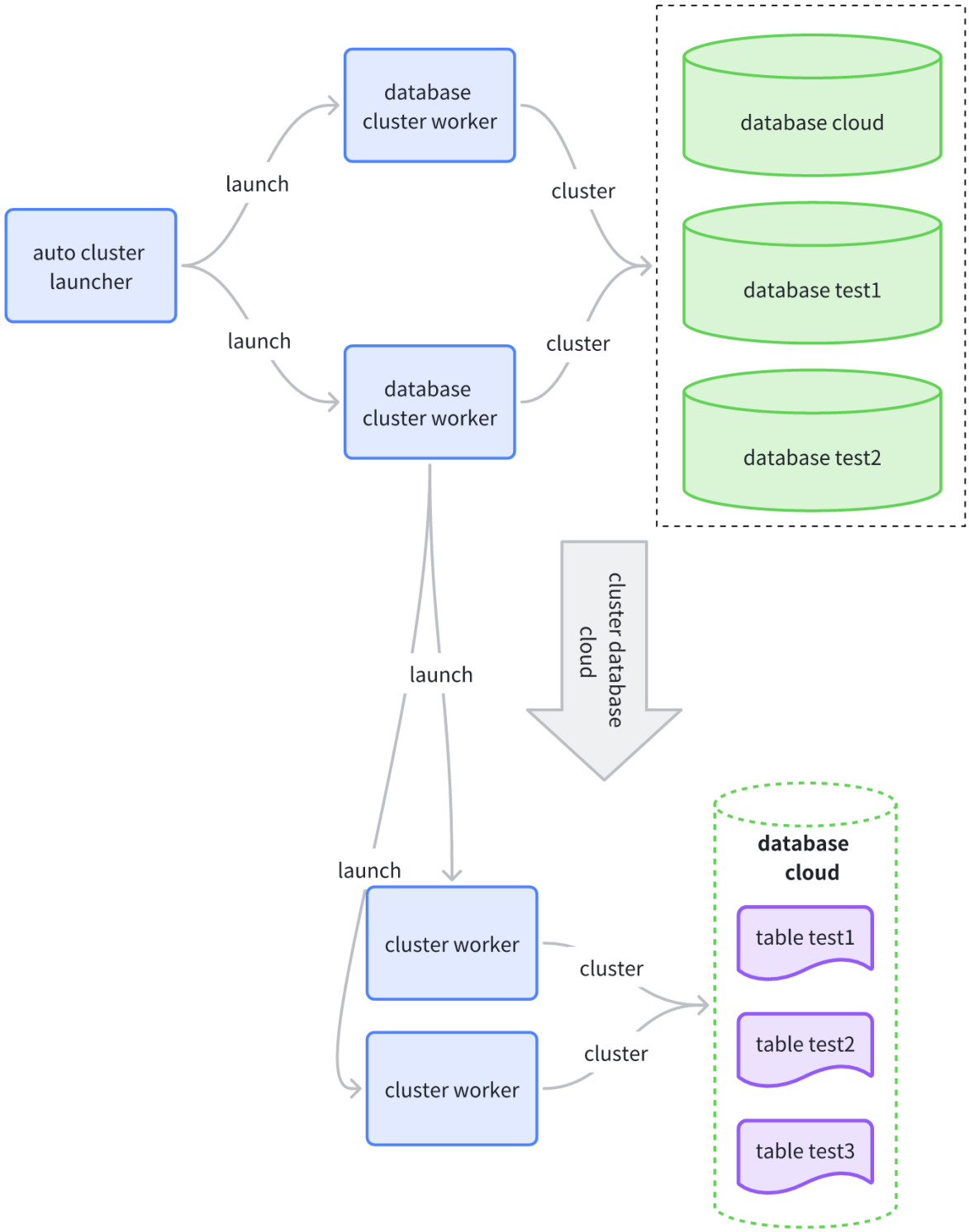
Parameter name	Description	Default value
`cloud.enable_auto_cluster`	(Master switch) Controls whether to enable the auto-clustering feature.	`ON`
`cloud.cluster_worker_launc`	The scheduling interval for the background service to check tables.	`30s`
`cloud.num_cluster_data_fil`	The threshold for the number of incremental data files that triggers a reordering operation on a table.	`10`
`cloud.max_database_cluste`	The maximum number of databases for which reordering tasks can be executed in parallel.	`8`
`cloud.max_table_cluster_wc`	The maximum number of tables within a single database for which reordering operations can be performed in parallel.	`2`

These parameters can be set at the session or system level:

```
-- Disables the auto-clustering feature at the system level.
ALTER SYSTEM SET cloud.enable_auto_cluster = OFF;

-- Adjusts the number of worker processes for parallel table processing.
SET cloud.max_table_cluster_workers = 4;

-- Adjusts the incremental file threshold for triggering physical reordering.
SET cloud.num_cluster_data_files = 20;
```

How it works

The workflow of the auto-clustering feature is shown in the diagram below.



The core components of this process include:

- **auto cluster launcher**: Wakes up periodically according to the interval set by

`cloud.cluster_worker_launch_interval` and is responsible for starting worker processes for databases.

- **database cluster worker**: The maximum number of these workers is controlled by `cloud.max_database_cluster_workers`. It scans all tables with a defined cluster key within a single database and sorts them based on the number of incremental data files, prioritizing tables with the most incremental files.
- **cluster worker (for tables)**: The maximum number of these workers is controlled by `cloud.max_table_cluster_workers`. When the number of a table's incremental data files exceeds the `cloud.num_cluster_data_files` threshold, this process executes a `CLUSTER` operation on the table.

Chapter 6

Manage System

6.1 Configure Database System

Server configuration parameters affect the behavior of SynxDB Elastic. They are part of the PostgreSQL "Grand Unified Configuration"system, so they are sometimes called "GUCs". Most of the SynxDB Elastic server configuration parameters are the same as the PostgreSQL configuration parameters, but some are specific to SynxDB Elastic.

6.1.1 Set parameters at the database level

Use `ALTER DATABASE` to set parameters at the database level. For example:

```
ALTER DATABASE mydatabase SET search_path TO myschema;
```

When you set a session parameter at the database level, every session that connects to that database uses that parameter setting.

6.1.2 Set parameters at the row level

Use `ALTER ROLE` to set a parameter at the role level. For example:

```
ALTER ROLE bob SET search_path TO bobschema;
```

When you set a session parameter at the role level, every session initiated by that role uses that

parameter setting. Settings at the role level override settings at the database level.

6.1.3 Set parameters in a session level

Any session parameter can be set in an active database session using the `SET` command. For example:

```
SET statement_mem TO '200MB';
```

The parameter setting is valid for the rest of that session or until you issue a `RESET` command. For example:

```
RESET statement_mem;
```

Settings at the session level override those at the role level.

6.1.4 View parameters using the `pg_settings` view

In addition, you can use the `pg_settings` view to check all parameter settings:

```
SELECT name, setting, unit, category, short_desc, context, vartype, min_val, max_val
FROM pg_settings
ORDER BY category, name;
```

Field descriptions:

- `name`: Parameter name.
- `setting`: Current parameter value.
- `unit`: Parameter unit (if applicable, such as `ms` and `kB`).
- `category`: The configuration category that the parameter belongs to.
- `short_desc`: Brief description of the parameter.
- `context`: Required context for parameter modification, possible values include:
  - `internal`: Used internally by the system, cannot be modified directly.
  - `postmaster`: Requires database instance restart to take effect.
  - `sighup`: Loads configuration via `pg_ctl reload` or `SELECT pg_reload_conf();`.

    - `superuser`: Only superusers can modify.
    - `user`: Regular users can modify.
- `vartype`: Parameter type, such as `bool`, `integer`, `real`, `string`.
- `min_val` and `max_val`: Minimum and maximum allowed values (if applicable).

Use cases:

- View parameters of a specific category:

```
SELECT name, setting, short_desc
FROM pg_settings
WHERE category = 'Resource Usage / Memory';
```

- Search for parameters containing specific keywords:

```
SELECT name, setting, short_desc
FROM pg_settings
WHERE name LIKE '%work_mem%';
```

- Check parameters requiring restart to take effect:

```
SELECT name, setting, context
FROM pg_settings
WHERE context = 'postmaster';
```

- Display parameters effective in the current database session:

```
SELECT name, setting
FROM pg_settings
WHERE context = 'user';
```

To change configuration parameters, you can use SQL commands, for example:

```
ALTER SYSTEM SET parameter_name = 'value';
```

6.2 Routine Maintenance Operations

This document describes routine maintenance operations that you can perform on SynxDB Elastic.

6.2.1 View the definition of an object

To see the definition of an object, such as a table or view, you can use the `\d+` meta-command when working in `psql`. For example, to see the definition of a table:

```
\d+ t2

                                        Table "public.t2"

Column |  Type   | Collation | Nullable | Default | Storage | Compression | Stats
target | Description
--------+---------+-----------+----------+---------+---------+-------------+----------
----+-------------
c1     | integer |           |          |         | plain   |             |
  |
c2     | integer |           |          |         | plain   |             |
  |
Tablespace: "default_tablespace"
```

6.2.2 View session memory usage information

You can create and use the `session_level_memory_consumption` view that provides information about the current memory utilization for sessions that are running queries on SynxDB Elastic. The view contains session information and information such as the database that the session is connected to, the query that the session is currently running, and memory consumed by the session processes.

- Create the `session_level_memory_consumption` view
- About the `session_level_memory_consumption` view

Create the `session_level_memory_consumption` view

To create the `session_state.session_level_memory_consumption` view in a SynxDB Elastic, run the command `CREATE EXTENSION gp_internal_tools;` once for each database. For example, to install the view in the database `testdb`, use this command:

```
CREATE EXTENSION gp_internal_tools;
```

About the `session_level_memory_consumption` view

The `session_state.session_level_memory_consumption` view provides information about memory consumption and idle time for sessions that are running SQL queries.

When resource queue-based resource management is active, the column `is_runaway` indicates whether SynxDB Elastic considers the session a runaway session based on the`vmem` memory consumption of the session's queries. Under the resource queue-based resource management scheme, SynxDB Elastic considers the session a runaway when the queries consume an excessive amount of memory. The SynxDB Elastic server configuration parameter `runaway_detector_activation_percent` controls the conditions under which SynxDB Elastic considers a session a runaway session.

6.2.3 View and log per-process memory usage information

SynxDB Elastic allocates all memory within memory contexts. Memory contexts are a convenient way to manage memory that needs to live for differing amounts of time. Destroying a context releases all of the memory that was allocated in it.

Tracking the amount of memory used by a server process or a long-running query can help detect the source of a potential out-of-memory condition. SynxDB Elastic provides a system view and administration functions that you can use for this purpose.

6.2.4 About the `pg_backend_memory_contexts` view

To display the memory usage of all active memory contexts in the server process attached to the current session, use the `pg_backend_memory_contexts` system view. This view is restricted to superusers, but access might be granted to other roles.

```
SELECT * FROM pg_backend_memory_contexts;
```

About the memory context admin functions

You can use the system administration function `pg_log_backend_memory_contexts()` to instruct SynxDB Elastic to dump the memory usage of other sessions running on the coordinator host into the server log. Execution of this function is restricted to superusers only, and cannot be granted to other roles.

The signature of `pg_log_backend_memory_contexts()` function follows:

```
pg_log_backend_memory_contexts( pid integer )
```

where `pid` identifies the process whose memory contexts you want dumped.

`pg_log_backend_memory_contexts()` returns `t` when memory context logging is successfully activated for the process on the local host. When logging is activated, SynxDB Elastic writes one message to the log for each memory context at the `LOG` message level. The log messages appear in the server log based on the log configuration set; refer to Error Reporting and Logging in the PostgreSQL documentation for more information. *The memory context log messages are not sent to the client.*

The command triggered the dumping of the following (subset of) memory context messages to the local server log file:

```
SELECT pg_log_backend_memory_contexts( pg_backend_pid() );
```

```
2025-03-20 16:45:57.228512 UTC,"gpadmin","testdb",p16389,th-557447104,"[local]",,2025-
03-20 15:57:32 UTC,0,,cmd10,seg-1,,,,sx1,"LOG","00000","logging memory contexts of PID
16389",,,,,,"SELECT pg_log_backend_memory_contexts(pg_backend_pid());",0,,"mcxt.c",
1278,
2025-03-20 16:45:57.229275 UTC,"gpadmin","testdb",p16389,th-557447104,"[local]",,2025-
03-20 15:57:32 UTC,0,,cmd10,seg-1,,,,sx1,"LOG","00000","level: 0; TopMemoryContext:
108384 total in 6 blocks; 23248 free (21 chunks); 85136 used",,,,,,,0,,"mcxt.c",884,
2025-03-20 16:45:57.229822 UTC,"gpadmin","testdb",p16389,th-557447104,"[local]",,2025-
03-20 15:57:32 UTC,0,,cmd10,seg-1,,,,sx1,"LOG","00000","level: 1; pgstat
TabStatusArray lookup hash table: 8192 total in 1 blocks; 1416 free (0 chunks); 6776
used",,,,,,,0,,"mcxt.c",884,
2025-03-20 16:45:57.230387 UTC,"gpadmin","testdb",p16389,th-557447104,"[local]",,2025-
03-20 15:57:32 UTC,0,,cmd10,seg-1,,,,sx1,"LOG","00000","level: 1;
TopTransactionContext: 8192 total in 1 blocks; 7576 free (1 chunks); 616 used",,,,,,,
0,,"mcxt.c",884,
2025-03-20 16:45:57.230961 UTC,"gpadmin","testdb",p16389,th-557447104,"[local]",,2025-
03-20 15:57:32 UTC,0,,cmd10,seg-1,,,,sx1,"LOG","00000","level: 1; TableSpace cache:
8192 total in 1 blocks; 2056 free (0 chunks); 6136 used",,,,,,,0,,"mcxt.c",884,
```

6.3 Backup and Restore

6.3.1 Backup and Restore Overview

SynxDB Elastic offers both parallel and non-parallel methods for database backups and restores. Parallel operations handle large systems efficiently because each segment host writes data to its local disk at the same time. Non-parallel operations, however, transfer all data over the network to the coordinator, which then writes it to its storage. This method not only concentrates I/O on a single host but also requires the coordinator to have enough local disk space for the entire database.

Parallel backup with `gpbackup` and `gprestore`

SynxDB Elastic provides `gpbackup` and `gprestore` for parallel backup and restore utilities. `gpbackup` uses table-level `ACCESS SHARE` locks instead of `EXCLUSIVE` locks on the `pg_class` catalog table. This enables you to execute DDL statements such as `CREATE`, `ALTER`, `DROP`, and `TRUNCATE` during backups, as long as these statements do not target the current backup set.

Backup files created with `gpbackup` are designed to provide future capabilities for restoring individual database objects along with their dependencies, such as functions and required user-defined data types.

For details about backup and restore using `gpbackup` and `gprestore`, see *Perform Full Backup and Restore* and *Perform Incremental Backup and Restore*.

Command-line flags for gpbackup and gprestore

The command-line flags for `gpbackup` are as follows:

```
Usage:
gpbackup [flags]

Flags:
      --backup-dir string            The absolute path of the directory to which all
backup files will be written
      --compression-level int        Level of compression to use during data backup.
Range of valid values depends on compression type (default 1)
      --compression-type string      Type of compression to use during data backup.
Valid values are 'gzip', 'zstd' (default "gzip")
      --copy-queue-size int          number of COPY commands gpbackup should enqueue
when backing up using the --single-data-file option (default 1)
```

(continues on next page)

(continued from previous page)

```
      --data-only                    Only back up data, do not back up metadata
      --dbname string                The database to be backed up
      --debug                        Print verbose and debug log messages
      --exclude-schema stringArray   Back up all metadata except objects in the
specified schema(s). --exclude-schema can be specified multiple times.
      --exclude-schema-file string   A file containing a list of schemas to be
excluded from the backup
      --exclude-table stringArray    Back up all metadata except the specified
table(s). --exclude-table can be specified multiple times.
      --exclude-table-file string    A file containing a list of fully-qualified
tables to be excluded from the backup
      --from-timestamp string        A timestamp to use to base the current
incremental backup off
      --help                         Help for gpbackup
      --include-schema stringArray   Back up only the specified schema(s). --include-
schema can be specified multiple times.
      --include-schema-file string   A file containing a list of schema(s) to be
included in the backup
      --include-table stringArray    Back up only the specified table(s). --include-
table can be specified multiple times.
      --include-table-file string    A file containing a list of fully-qualified
tables to be included in the backup
      --incremental                  Only back up data for AO tables that have been
modified since the last backup
      --jobs int                     The number of parallel connections to use when
backing up data (default 1)
      --leaf-partition-data          For partition tables, create one data file per
leaf partition instead of one data file for the whole table
      --metadata-only                Only back up metadata, do not back up data
      --no-compression               Skip compression of data files
      --plugin-config string         The configuration file to use for a plugin
      --quiet                        Suppress non-warning, non-error log messages
      --single-data-file             Back up all data to a single file instead of one
per table
      --verbose                      Print verbose log messages
      --version                      Print version number and exit
      --with-stats                   Back up query plan statistics
      --without-globals              Skip backup of global metadata
```

The command-line flags for `gprestore` are as follows:

```
Usage:
gprestore [flags]

Flags:
      --backup-dir string            The absolute path of the directory in which the
backup files to be restored are located
      --copy-queue-size int          Number of COPY commands gprestore should enqueue
when restoring a backup taken using the --single-data-file option (default 1)
      --create-db                    Create the database before metadata restore
      --data-only                    Only restore data, do not restore metadata
      --debug                        Print verbose and debug log messages
      --exclude-schema stringArray   Restore all metadata except objects in the
specified schema(s). --exclude-schema can be specified multiple times.
      --exclude-schema-file string   A file containing a list of schemas that will not
be restored
      --exclude-table stringArray    Restore all metadata except the specified
relation(s). --exclude-table can be specified multiple times.
      --exclude-table-file string    A file containing a list of fully-qualified
relation(s) that will not be restored
      --help                         Help for gprestore
      --include-schema stringArray   Restore only the specified schema(s). --include-
schema can be specified multiple times.
      --include-schema-file string   A file containing a list of schemas that will be
restored
      --include-table stringArray    Restore only the specified relation(s). --
include-table can be specified multiple times.
      --include-table-file string    A file containing a list of fully-qualified
relation(s) that will be restored
      --incremental                  BETA FEATURE: Only restore data for all heap
tables and only AO tables that have been modified since the last backup
      --jobs int                     Number of parallel connections to use when
restoring table data and post-data (default 1)
      --metadata-only                Only restore metadata, do not restore data
      --on-error-continue            Log errors and continue restore, instead of
exiting on first error
      --plugin-config string         The configuration file to use for a plugin
      --quiet                        Suppress non-warning, non-error log messages
      --redirect-db string           Restore to the specified database instead of the
database that was backed up
      --redirect-schema string       Restore to the specified schema instead of the
```

(continues on next page)

(continued from previous page)

```
schema that was backed up
      --resize-cluster               Restore a backup taken on a cluster with more or
fewer segments than the cluster to which it will be restored
      --run-analyze                  Run ANALYZE on restored tables
      --timestamp string             The timestamp to be restored, in the format
YYYYMMDDHHMMSS
      --truncate-table               Removes data of the tables getting restored
      --verbose                      Print verbose log messages
      --version                      Print version number and exit
      --with-globals                 Restore global metadata
      --with-stats                   Restore query plan statistics
```

Non-parallel backup with `pg_dump`

You can also use the PostgreSQL non-parallel backup utilitiesm`pg_dump` and `pg_dumpall` to create a single dump file on the coordinator host that contains all data from all active segments.

The PostgreSQL non-parallel utilities should be used only for special cases. They are much slower than using `gpbackup` and `gprestore` because all of the data must pass through the coordinator. In addition, it is often the case that the coordinator host has insufficient disk space to save a backup of an entire distributed SynxDB Elastic.

The `pg_restore` utility requires compressed dump files created by `pg_dump` or `pg_dumpall`. Before starting the restore, you should modify the `CREATE TABLE` statements in the dump files to include the SynxDB Elastic `DISTRIBUTED` clause. If you do not include the `DISTRIBUTED` clause, SynxDB Elastic assigns default values, which might not be optimal.

To perform a non-parallel restore using parallel backup files, you can copy the backup files from each segment host to the coordinator host, and then load them through the coordinator.

Another non-parallel method for backing up SynxDB Elastic data is to use the `COPY TO` SQL command to copy all or a portion of a table out of the database to a delimited text file on the coordinator host.

Backup support for UnionStore tables

SynxDB Elastic supports backing up UnionStore tables using the standard PostgreSQL `pg_dump` utility. UnionStore tables can be backed up and restored just like regular heap tables, maintaining their storage format and access method.

When using `pg_dump` with UnionStore tables:

- The table structure and data are exported in the standard PostgreSQL format
- The `USING union_store` clause is preserved in the dump file
- The restore process will recreate the tables with the correct UnionStore access method

Example of backing up a UnionStore table:

```
-- Create a UnionStore table
CREATE TABLE unionstore_table (id int, name text) TABLESPACE ts_union_store;
INSERT INTO unionstore_table VALUES (1, 'test');

-- Backup using pg_dump
pg_dump -t unionstore_table mydatabase > unionstore_backup.sql

-- Restore the table
psql mydatabase < unionstore_backup.sql
```

Note

- UnionStore tables are fully compatible with `pg_dump`
- The backup process preserves the UnionStore access method and all associated metadata
- When restoring, ensure the UnionStore extension is installed in the target database

6.3.2 Perform Full Backup and Restore

SynxDB Elastic supports backing up and restoring the full database in parallel. Parallel operations scale regardless of the number of segments in your system, because segment hosts each write their data to local disk storage at the same time.

`gpbackup` and `gprestore` are SynxDB Elastic command-line utilities that create and restore backup sets for SynxDB Elastic. By default, `gpbackup` stores only the object metadata files and DDL files for a backup in the SynxDB Elastic coordinator data directory. SynxDB Elastic segments use the `COPY ... ON SEGMENT` command to store their data for backed-up tables in compressed CSV data files, located in each segment's backups directory.

The backup metadata files contain all of the information that `gprestore` needs to restore a full backup set in parallel. Each `gpbackup` task uses a single transaction in SynxDB Elastic. During this transaction, metadata is backed up on the coordinator host, and data for each table on each segment host is written to CSV backup files using `COPY ... ON SEGMENT` commands in parallel. The backup process acquires an `ACCESS SHARE` lock on each table that is backed up.

Back up the full database

To perform a complete backup of a database, as well as SynxDB Elastic system metadata, use the command:

```
gpbackup --dbname <database_name>
```

For example:

```
$ gpbackup --dbname test_04

20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-gpbackup version =
1.2.7-beta1+dev.7
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-|product_name|
Version = oudberry Database 1.0.0 build 5551471267
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Starting backup of
database test_04
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Backup Timestamp =
20240108171718
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Backup Database =
test_04
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Gathering table
```

(continues on next page)

(continued from previous page)

```
state information
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Acquiring ACCESS
SHARE locks on tables
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Gathering
additional table metadata
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Getting storage
information
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[WARNING]:-No tables in
backup set contain data. Performing metadata-only backup instead.
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Metadata will be
written to /data0/coordinator/gpseg-1/backups/20240108/20240108171718/gpbackup_
20240108171718_metadata.sql
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Writing global
database metadata
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Global database
metadata backup complete
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Writing pre-data
metadata
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Pre-data metadata
metadata backup complete
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Writing post-data
metadata
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Post-data metadata
backup complete
20240108:17:17:19 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Found neither /usr/
local/cloudberry-db-1.0.0/bin/gp_email_contacts.yaml nor /home/gpadmin//gp_email_
contacts.yaml
20240108:17:17:19 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Email containing
gpbackup report /data0/coordinator/gpseg-1/backups/20240108/20240108171718/gpbackup_
20240108171718_report will not be sent
20240108:17:17:19 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Backup completed
successfully
```

The above command creates a file that contains global and database-specific metadata on the SynxDB Elastic coordinator host in the default directory, `$COORDINATOR_DATA_DIRECTORY/backups/<YYYYMMDD>/<YYYYMMDDHHMMSS>/`. For example:

```
ls $COORDINATOR_DATA_DIRECTORY/backups/20240108/20240108171718

gpbackup_20240108171718_config.yaml   gpbackup_20240108171718_report
gpbackup_20240108171718_metadata.sql  gpbackup_20240108171718_toc.yaml
```

By default, each segment stores each table's data for the backup in a separate compressed CSV file in `<seg_dir>/backups/<YYYYMMDD>/<YYYYMMDDHHMMSS>/`. For example:

```
ls /data1/primary/gpseg1/backups/20240108/20240108171718/

gpbackup_0_20240108171718_17166.gz  gpbackup_0_20240108171718_26303.gz
gpbackup_0_20240108171718_21816.gz
```

To consolidate all backup files into a single directory, include the `--backup-dir` option. Note that you need to specify an absolute path with this option:

```
$ gpbackup --dbname test_04 --backup-dir /home/gpadmin/backups

20240108:17:34:10 gpbackup:gpadmin:cbdb-coordinator:003348-[INFO]:-gpbackup version =
1.2.7-beta1+dev.7
20240108:17:34:10 gpbackup:gpadmin:cbdb-coordinator:003348-[INFO]:-|product_name|
Version = oudberry Database 1.0.0 build 5551471267
...
20240108:17:34:12 gpbackup:gpadmin:cbdb-coordinator:003348-[INFO]:-Backup completed
successfully

$ find /home/gpadmin/backups/ -type f

/home/gpadmin/backups/gpseg0/backups/20240108/20240108173410/gpbackup_0_
20240108173410_16593.gz
/home/gpadmin/backups/gpseg-1/backups/20240108/20240108173410/gpbackup_20240108173410_
config.yaml
/home/gpadmin/backups/gpseg-1/backups/20240108/20240108173410/gpbackup_20240108173410_
report
/home/gpadmin/backups/gpseg-1/backups/20240108/20240108173410/gpbackup_20240108173410_
toc.yaml
/home/gpadmin/backups/gpseg-1/backups/20240108/20240108173410/gpbackup_20240108173410_
metadata.sql
/home/gpadmin/backups/gpseg1/backups/20240108/20240108173410/gpbackup_1_
20240108173410_16593.gz
```

When performing a backup operation, you can use the `--single-data-file` in situations where the additional overhead of multiple files might be prohibitive. For example, if you use a third party storage solution such as Data Domain with backups.

> **Tip**
>
> Backing up a materialized view does not back up the materialized view data. Only the materialized view definition is backed up.

Restore the full database

To use `gprestore` to restore from a backup set, you must use the `--timestamp` option to specify the exact timestamp value (`YYYYMMDDHHMMSS`) to restore. Include the `--create-db` option if the database does not exist in the cluster. For example:

```
$ dropdb demo
$ gprestore --timestamp 20240108171718 --create-db

20240108:17:42:26 gprestore:gpadmin:cbdb-coordinator:004115-[INFO]:-Restore Key =
20240108171718
20240108:17:42:26 gprestore:gpadmin:cbdb-coordinator:004115-[INFO]:-gpbackup version =
1.2.7-beta1+dev.7
20240108:17:42:26 gprestore:gpadmin:cbdb-coordinator:004115-[INFO]:-gprestore version
= 1.2.7-beta1+dev.7
20240108:17:42:26 gprestore:gpadmin:cbdb-coordinator:004115-[INFO]:-|product_name|
Version = oudberry Database 1.0.0 build 5551471267
20240108:17:42:26 gprestore:gpadmin:cbdb-coordinator:004115-[INFO]:-Creating database
20240108:17:42:26 gprestore:gpadmin:cbdb-coordinator:004115-[INFO]:-Database creation
complete for: test_04
20240108:17:42:26 gprestore:gpadmin:cbdb-coordinator:004115-[INFO]:-Restoring pre-data
metadata
Pre-data objects restored:  3 / 3 [================================] 100.00% 0s
20240108:17:42:26 gprestore:gpadmin:cbdb-coordinator:004115-[INFO]:-Pre-data metadata
restore complete
20240108:17:42:26 gprestore:gpadmin:cbdb-coordinator:004115-[INFO]:-Restoring post-
data metadata
20240108:17:42:26 gprestore:gpadmin:cbdb-coordinator:004115-[INFO]:-Post-data metadata
restore complete
20240108:17:42:26 gprestore:gpadmin:cbdb-coordinator:004115-[INFO]:-Found neither /
```

(continues on next page)

(continued from previous page)

```
usr/local/cloudberry-db-1.0.0/bin/gp_email_contacts.yaml nor /home/gpadmin//gp_email_
contacts.yaml
20240108:17:42:26 gprestore:gpadmin:cbdb-coordinator:004115-[INFO]:-Email containing
gprestore report /data0/coordinator/gpseg-1/backups/20240108/20240108171718/gprestore_
20240108171718_20240108174226_report will not be sent
20240108:17:42:26 gprestore:gpadmin:cbdb-coordinator:004115-[INFO]:-Restore completed
successfully
```

If you specified a custom `--backup-dir` to consolidate the backup files, include the same `--backup-dir` option when using `gprestore` to locate the backup files:

```
$ dropdb test_04
$ gprestore --backup-dir /home/gpadmin/backups/ --timestamp 20240109102646 --create-db

20240109:10:33:17 gprestore:gpadmin:cbdb-coordinator:017112-[INFO]:-Restore Key =
20240109102646
...
20240109:10:33:17 gprestore:gpadmin:cbdb-coordinator:017112-[INFO]:-Restore completed
successfully
```

`gprestore` does not attempt to restore global metadata for the SynxDB Elastic system by default. If this is required, include the `--with-globals` argument.

By default, `gprestore` uses 1 connection to restore table data and metadata. If you have a large backup set, you can improve performance of the restore by increasing the number of parallel connections with the `--jobs` option. For example:

```
$ gprestore --backup-dir /home/gpadmin/backups/ --timestamp 20240109102646 --create-db
--jobs 4
```

Test the number of parallel connections with your backup set to determine the ideal number for fast recovery.

Tip

You cannot perform a parallel restore operation with `gprestore` if the backup combines table backups into a single file per segment with the gpbackup option `--single-data-file`.

Restoring a materialized view does not restore materialized view data. Only the materialized view definition is restored. To populate the materialized view with data, use `REFRESH MATERIALIZED VIEW`. When you refresh the materialized view, the tables that are referenced by the materialized view definition must be available. The `gprestore` log file lists the materialized views that have been restored and the `REFRESH MATERIALIZED VIEW` commands that are used to populate the materialized views with data.

Filter the contents of a backup or restore

Filter by schema

`gpbackup` backs up all schemas and tables in the specified database, unless you exclude or include individual schema or table objects with schema level or table level filter options.

The schema level options are `--include-schema`, `--include-schema-file`, or `--exclude-schema`, `--exclude-schema-file` command-line options to gpbackup. For example, if the `test_04` database includes only 2 schemas, `schema1` and `schema2`, both of the following commands back up only the `schema1` schema:

```
$ gpbackup --dbname test_04 --include-schema schema1
$ gpbackup --dbname test_04 --exclude-schema schema2
```

You can include multiple `--include-schema` options in a `gpbackup` or multiple `--exclude-schema` options. For example:

```
$ gpbackup --dbname test_04 --include-schema schema1 --include-schema schema2
```

If you have a large number of schemas, you can list the schemas in a text file and specify the file with the `--include-schema-file` or `--exclude-schema-file` options in a `gpbackup` command. Each line in the file must define a single schema, and the file cannot contain trailing lines. For example, this command uses a file in the `gpadmin` home directory to include a set of schemas.

```
$ gpbackup --dbname test_04 --include-schema-file /home/gpadmin/backup-schemas.txt --
backup-dir /home/gpadmin/backups
```

Filter by table

To filter the individual tables that are included in a backup set, or excluded from a backup set, specify individual tables with the `--include-table` option or the `--exclude-table` option. The table must be schema qualified, `<schema-name>.<table-name>`. The individual table filtering options can be specified multiple times. However, `--include-table` and `--exclude-table` cannot both be used in the same command.

Tip

If you have used the `--include-table` option in a `gpbackup` command, the database, schema, and the sequence related to the target table you specified are not backed up. Before you restore the backup set, you must create the database, schema, and sequence manually.

For example, if you have used `gpbackup --dbname test_04 --include-table schema1.table1 --backup-dir /home/gpadmin/backups` to back up the `test_04` database, you must create the `test_04` database, `schema1` schema, and `schema1.table1_id_seq` sequence manually before you restore the backup set. Otherwise, the restore operation fails with the error message indicating that the database, schema, or sequence does not exist.

If a table or schema name uses any character other than a lowercase letter, number, or an underscore character, then you must include that name in double quotes. For example:

```
gpbackup --dbname test1 --include-table "schema1"."ComplexName Table" --backup-dir /
home/gpadmin/backups
```

You can create a list of qualified table names in a text file. When listing tables in a file, each line in the text file must define a single table using the format `<schema-name>.<table-name>`. The file must not include trailing lines or double quotes even if table or schema name uses any character other than a lowercase letter, number, or an underscore character. For example:

```
schema1.table1
schema2.table2
schema1.ComplexName Table
```

After creating the file, you can use it either to include or exclude tables with the gpbackup options `--include-table-file` or `--exclude-table-file`. For example:

```
$ gpbackup --dbname test_04 --include-table-file /home/gpadmin/table-list.txt
```

You can combine `--include schema` with `--exclude-table` or `--exclude-table-file` for a backup. The following example uses `--include-schema` with `--exclude-table` to back up a schema except for a single table.

```
$ gpbackup --dbname test_04 --include-schema schema1 --exclude-table schema2.table2
```

You cannot combine `--include-schema` with `--include-table` or `--include-table-file`, and you cannot combine `--exclude-schema` with any table filtering option such as `--exclude-table` or `--include-table`.

When you use `--include-table` or `--include-table-file` dependent objects are not automatically backed up or restored, you must explicitly specify the dependent objects that are required. For example, if you back up or restore a view or materialized view, you must also specify the tables that the view or the materialized view uses. If you backup or restore a table that uses a sequence, you must also specify the sequence.

Filter with `gprestore`

After creating a backup set with `gpbackup`, you can filter the schemas and tables that you want to restore from the backup set using the `gprestore --include-schema` and `--include-table-file` options. These options work in the same way as their `gpbackup` counterparts, but have the following restrictions:

- The tables that you attempt to restore must not already exist in the database.
- If you attempt to restore a schema or table that does not exist in the backup set, the `gprestore` does not execute.
- If you use the `--include-schema` option, `gprestore` cannot restore objects that have dependencies on multiple schemas.
- If you use the `--include-table-file` option, `gprestore` does not create roles or set the owner of the tables. The utility restores table rules. Triggers are also restored but are not supported in SynxDB Elastic.
- The file that you specify with `--include-table-file` cannot include a leaf partition name, as it can when you specify this option with `gpbackup`. If you specified leaf partitions in the

backup set, specify the partitioned table to restore the leaf partition data.

When restoring a backup set that contains data from some leaf partitions of a partitioned table, the partitioned table is restored along with the data for the leaf partitions. For example, you create a backup with the `gpbackup` option `--include-table-file` and the text file lists some leaf partitions of a partitioned table. Restoring the backup creates the partitioned table and restores the data only for the leaf partitions listed in the file.

Filter by leaf partition

By default, `gpbackup` creates one file for each table on a segment. You can specify the `--leaf-partition-data` option to create one data file per leaf partition of a partitioned table, instead of a single file. You can also filter backups to specific leaf partitions by listing the leaf partition names in a text file to include. For example, consider a table that was created using the statement:

```
CREATE TABLE sales (id int, date date, amt decimal(10,2))
PARTITION BY RANGE (date)
( PARTITION Jan23 START (date '2023-01-01') INCLUSIVE ,
  PARTITION Feb23 START (date '2023-02-01') INCLUSIVE ,
  PARTITION Mar23 START (date '2023-03-01') INCLUSIVE ,
  PARTITION Apr23 START (date '2023-04-01') INCLUSIVE ,
  PARTITION May23 START (date '2023-05-01') INCLUSIVE ,
  PARTITION Jun23 START (date '2023-06-01') INCLUSIVE ,
  PARTITION Jul23 START (date '2023-07-01') INCLUSIVE ,
  PARTITION Aug23 START (date '2023-08-01') INCLUSIVE ,
  PARTITION Sep23 START (date '2023-09-01') INCLUSIVE ,
  PARTITION Oct23 START (date '2023-10-01') INCLUSIVE ,
  PARTITION Nov23 START (date '2023-11-01') INCLUSIVE ,
  PARTITION Dec23 START (date '2023-12-01') INCLUSIVE
  END (date '2024-01-01') EXCLUSIVE );

NOTICE:  CREATE TABLE will create partition "sales_1_prt_jan23" for table "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_feb23" for table "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_mar23" for table "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_apr23" for table "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_may23" for table "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_jun23" for table "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_jul23" for table "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_aug23" for table "sales"
```

(continues on next page)

(continued from previous page)

```
NOTICE:  CREATE TABLE will create partition "sales_1_prt_sep23" for table "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_oct23" for table "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_nov23" for table "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_dec23" for table "sales"
CREATE TABLE
```

To back up only data for the last quarter of the year, first create a text file that lists those leaf partition names instead of the full table name:

```
public.sales_1_prt_oct23
public.sales_1_prt_nov23
public.sales_1_prt_dec23
```

Then specify the file with the `--include-table-file` option to generate one data file per leaf partition:

```
$ gpbackup --dbname test_04 --include-table-file last-quarter.txt --leaf-partition-
data
```

When you specify `--leaf-partition-data`, `gpbackup` generates one data file per leaf partition when backing up a partitioned table. For example, this command generates one data file for each leaf partition:

```
$ gpbackup --dbname test_04 --include-table public.sales --leaf-partition-data
```

When leaf partitions are backed up, the leaf partition data is backed up along with the metadata for the entire partitioned table.

Check report files

When performing a backup or restore operation, `gpbackup` and `gprestore` generate a report file that contains the detailed information of the operations. When email notification is configured, the email sent contains the contents of the report file. For information about email notification, see *Configure email notifications*.

The report file is placed in the SynxDB Elastic coordinator backup directory. The report file name contains the timestamp of the operation. Thes following are the formats of the `gpbackup` and `gprestore` report file names.

```
gpbackup_<backup_timestamp>_report
gprestore_<backup_timestamp>_<restore_timesamp>_report
```

For these example report file names, `20240109111719` is the timestamp of the backup and `20240109112545` is the timestamp of the restore operation.

```
gpbackup_20240109111719_report
gprestore_20240109111719_20240109112545_report
```

This backup directory on a SynxDB Elastic coordinator host contains both a `gpbackup` and `gprestore` report file.

```
$ ls -l /data0/coordinator/gpseg-1/backups/20240109/20240109111719/
total 24

-r--r--r-- 1 gpadmin gpadmin  715 Jan  9 11:17 gpbackup_20240109111719_config.yaml
-r--r--r-- 1 gpadmin gpadmin  895 Jan  9 11:17 gpbackup_20240109111719_metadata.sql
-r--r--r-- 1 gpadmin gpadmin 1441 Jan  9 11:17 gpbackup_20240109111719_report
-r--r--r-- 1 gpadmin gpadmin 1226 Jan  9 11:17 gpbackup_20240109111719_toc.yaml
-r--r--r-- 1 gpadmin gpadmin  569 Jan  9 11:21 gprestore_20240109111719_
20240109112128_report
-r--r--r-- 1 gpadmin gpadmin  514 Jan  9 11:25 gprestore_20240109111719_
20240109112545_report
```

The contents of the report files are similar. This is an example of the contents of a `gprestore` report file.

```
|product_name| Restore Report

timestamp key:           20240109111719
gpdb version:            oudberry Database 1.0.0 build 5551471267
gprestore version:       1.2.7-beta1+dev.7

database name:           test_04
command line:            gprestore --timestamp 20240109111719 --create-db

backup segment count:    2
restore segment count:   2
start time:              Tue Jan 09 2024 11:25:45
```

(continues on next page)

(continued from previous page)

```
end time:                Tue Jan 09 2024 11:25:46
duration:                0:00:01

restore status:          Success
```

Configure email notifications

`gpbackup` and `gprestore` can send email notifications after a back up or restore operation completes.

To have `gpbackup` or `gprestore` send out status email notifications, you need place a file named `gp_email_contacts.yaml` in the home directory of the user running `gpbackup` or `gprestore` in the same directory as the utilities (`$GPHOME/bin`). A utility issues a message if it cannot locate a `gp_email_contacts.yaml` file in either location. If both locations contain a `.yaml` file, the utility uses the file in user `$HOME`.

The email subject line includes the utility name, timestamp, job status (Success or Failure), and the name of the SynxDB Elastic host `gpbackup` or `gprestore` is called from. These are example subject lines for `gpbackup` emails.

```
gpbackup 20180202133601 on gp-master completed: Success
```

or

```
gpbackup 20200925140738 on mdw completed: Failure
```

The email contains summary information about the operation including options, duration, and number of objects backed up or restored. For information about the contents of a notification email, see Report Files.

> **Tip**
>
> The UNIX mail utility must be running on the SynxDB Elastic host and must be configured to allow the SynxDB Elastic superuser (`gpadmin`) to send email. Also ensure that the mail program executable is locatable via the `gpadmin` user's `$PATH`.

gpbackup and gprestore email file format

The `gpbackup` and `gprestore` email notification YAML file `gp_email_contacts.yaml` uses indentation (spaces) to determine the document hierarchy and the relationships of the sections to one another. The use of white space is significant. White space should not be used simply for formatting purposes, and tabs should not be used at all.

If the status parameters are not specified correctly, the utility does not issue a warning. For example, if the success parameter is misspelled and is set to true, a warning is not issued and an email is not sent to the email address after a successful operation. To ensure email notification is configured correctly, run tests with email notifications configured.

This is the format of the `gp_email_contacts.yaml` YAML file for gpbackup email notifications:

```
contacts:
  gpbackup:
  - address: <user>@<domain>
    status:
         success: [true | false]
         success_with_errors: [true | false]
         failure: [true | false]
  gprestore:
  - address: <user>@<domain>
    status:
         success: [true | false]
         success_with_errors: [true | false]
         failure: [true | false]
```

Email YAML file sections

`contacts`

Required. The section that contains the gpbackup and gprestore sections. The YAML file can contain a gpbackup section, a gprestore section, or one of each.

`gpbackup`

Optional. Begins the gpbackup email section.

`address`

Required. At least one email address must be specified. Multiple email address parameters can be specified. Each address requires a status section. `user@domain` is a single, valid email address.

`status`

Required. Specify when the utility sends an email to the specified email address. The default is to not send email notification. You specify sending email notifications based on the completion status of a backup or restore operation. At least one of these parameters must be specified and each parameter can appear at most once.

`success`

Optional. Specify whether an email is sent if the operation completes without errors. If the value is `true`, an email is sent if the operation completes without errors. If the value is `false` (the default), an email is not sent.

`success_with_errors`

Optional. Specify whether an email is sent if the operation completes with errors. If the value is `true`, an email is sent if the operation completes with errors. If the value is `false` (the default), an email is not sent.

`failure`

Optional. Specify if an email is sent if the operation fails. If the value is `true`, an email is sent if the operation fails. If the value is `false` (the default), an email is not sent.

`gprestore`

Optional. Begins the gprestore email section. This section contains the address and status parameters that are used to send an email notification after a gprestore operation. The syntax is the same as the gpbackup section.

Email examples

This example YAML file specifies sending email to email addresses depending on the success or failure of an operation. For a backup operation, an email is sent to a different address depending on the success or failure of the backup operation. For a restore operation, an email is sent to `gpadmin@example.com` only when the operation succeeds or completes with errors.

```
contacts:
  gpbackup:
  - address: gpadmin@example.com
    status:
      success: true
      success_with_errors: true
      failure: true
  gprestore:
  - address: gpadmin@example.com
    status:
      success: true
      success_with_errors: true
      failure: true
```

6.3.3 Perform Incremental Backup and Restore

Before reading this document, you are expected to first read the *Perform Full Backup and Restore* document.

To back up and restore tables incrementally, use the `gpbackup` and `gprestore` utilities. Incremental backups include all specified heap tables, and append-optimized tables (including column-oriented ones) that have changed. Even a single row change triggers a backup of the entire append-optimized table. For partitioned append-optimized tables, only the modified leaf partitions are backed up.

Incremental backups are efficient when the total amount of data in append-optimized tables or table partitions that changed is small compared to the data that has not changed since the last backup.

An incremental backup backs up an append-optimized table only if one of the following operations have been performed on the table after the last full or incremental backup:

- `ALTER TABLE`
- `DELETE`
- `INSERT`
- `TRUNCATE`
- `UPDATE`
- `DROP` and then re-create the table

To restore data from incremental backups, you need a complete incremental backup set.

About incremental backup sets

An incremental backup set includes the following backups:

- A full backup. This is the full backup that the incremental backups are based on.
- The set of incremental backups that capture the changes to the database from the time of the full backup.

For example, you can create a full backup and then create 3 daily incremental backups. The full backup and all 3 incremental backups are the backup set. For information about using an incremental backup set, see *Example using incremental backup sets*.

When you create or add to an incremental backup set, `gpbackup` ensures that the backups in the set are created with a consistent set of backup options to ensure that the backup set can be used in a restore operation. For information about backup set consistency, see *Use incremental backups*.

When you create an incremental backup you include these options with the other `gpbackup` options to create a backup:

- `--leaf-partition-data`: required for all backups in the incremental backup set.
    - Required when you create a full backup that will be the base backup for an incremental backup set.
    - Required when you create an incremental backup.
- `--incremental`: required when you create an incremental backup.

  You cannot combine `--data-only` or `--metadata-only` with `--incremental`.
- `--from-timestamp`: optional. This option can be used with `--incremental`. The timestamp you specify is an existing backup. The timestamp can be either a full backup or incremental backup. The backup being created must be compatible with the backup specified with the `--from-timestamp` option.

Use incremental backups

When you add an incremental backup to a backup set, `gpbackup` ensures that the full backup and the incremental backups are consistent by checking these `gpbackup` options:

- `--dbname`: the database must be the same.
- `--backup-dir`: the directory must be the same. The backup set, the full backup and the incremental backups, must be in the same location.
- `--single-data-file`: this option must be either specified or absent for all backups in the set.
- `--include-table-file`, `--include-schema`, or any other options that filter tables and schemas must be the same. When checking schema filters, only the schema names are checked, not the objects contained in the schemas.
- `--no-compression`: if this option is specified, it must be specified for all backups in the backup set.

If compression is used on the on the full backup, compression must be used on the incremental backups. Different compression levels are allowed for the backups in the backup set. For a backup, the default is compression level 1.

If you try to add an incremental backup to a backup set, the backup operation fails if the `gpbackup` options are not consistent.

Example using incremental backup sets

Each backup has a timestamp taken when the backup is created. For example, if you create a backup on May 14, 2023, the backup file names contain `20230514hhmmss`. The `hhmmss` represents the time: hour, minute, and second.

This example assumes that you have created two full backups and incremental backups of the database `mytest`. To create the full backups, you used this command:

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data
```

You created incremental backups with this command:

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data --incremental
```

When you specify the `--backup-dir` option, the backups are created in the `/mybackup` directory on each SynxDB Elastic host.

In the example, the full backups have the timestamp keys `20230514054532` and `20231114064330`. The other backups are incremental backups. The example consists of two backup sets, the first with two incremental backups, and second with one incremental backup. The backups are listed from earliest to most recent.

- `20230514054532` (full backup)
- `20230714095512`
- `20230914081205`
- `20231114064330` (full backup)
- `20230114051246`

To create a new incremental backup based on the latest incremental backup, you must include the same `--backup-dir` option as the incremental backup as well as the options

`--leaf-partition-data` and `--incremental`.

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data --incremental
```

You can specify the `--from-timestamp` option to create an incremental backup based on an existing incremental or full backup. Based on the example, this command adds a fourth incremental backup to the backup set that includes `20230914081205` as an incremental backup and uses `20230514054532` as the full backup.

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data --incremental --
from-timestamp 20230914081205
```

This command creates an incremental backup set based on the full backup `20231114064330` and is separate from the backup set that includes the incremental backup `20230114051246`.

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data --incremental --
from-timestamp 20231114064330
```

To restore a database with the incremental backup `20230914081205`, you need the incremental backups `20120914081205` and `20230714095512`, and the full backup `20230514054532`. This would be the `gprestore` command.

```
gprestore --backup-dir /backupdir --timestamp 20230914081205
```

Create an incremental backup with `gpbackup`

The `gpbackup` output displays the timestamp of the backup on which the incremental backup is based. In this example, the incremental backup is based on the backup with timestamp `20230802171642`. The backup `20230802171642` can be an incremental or full backup.

```
$ gpbackup --dbname test --backup-dir /backups --leaf-partition-data --incremental

20230803:15:40:51 gpbackup:gpadmin:mdw:002907-[INFO]:-Starting backup of database test
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Backup Timestamp =
20230803154051
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Backup Database = test
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Gathering list of tables for
backup
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Acquiring ACCESS SHARE locks on
```

(continues on next page)

(continued from previous page)

```
tables
Locks acquired:  5 / 5
[============================================================] 100.00% 0s
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Gathering additional table
metadata
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Metadata will be written to /
backups/gpseg-1/backups/20230803/20230803154051/gpbackup_20230803154051_metadata.sql
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Writing global database metadata
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Global database metadata backup
complete
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Writing pre-data metadata
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Pre-data metadata backup
complete
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Writing post-data metadata
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Post-data metadata backup
complete
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Basing incremental backup off of
backup with timestamp = 20230802171642
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Writing data to file
Tables backed up:  4 / 4
[============================================================] 100.00% 0s
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Data backup complete
20230803:15:40:53 gpbackup:gpadmin:mdw:002907-[INFO]:-Found neither /usr/local/
greenplum-db/./bin/gp_email_contacts.yaml nor /home/gpadmin/gp_email_contacts.yaml
20230803:15:40:53 gpbackup:gpadmin:mdw:002907-[INFO]:-Email containing gpbackup report
/backups/gpseg-1/backups/20230803/20230803154051/gpbackup_20230803154051_report will
not be sent
20230803:15:40:53 gpbackup:gpadmin:mdw:002907-[INFO]:-Backup completed successfully
```

Restore from an incremental backup with `gprestore`

When restoring an from an incremental backup, you can specify the `--verbose` option to display the backups that are used in the restore operation on the command line. For example, the following `gprestore` command restores a backup using the timestamp `20230807092740`, an incremental backup. The output includes the backups that were used to restore the database data.

```
$ gprestore --create-db --timestamp 20230807162904 --verbose
...
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[INFO]:-Pre-data metadata restore
```

(continues on next page)

(continued from previous page)

```
complete
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Verifying backup file count
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Restoring data from backup
with timestamp: 20230807162654
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for table public.
tbl_ao from file (table 1 of 1)
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Checking whether segment
agents had errors during restore
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Restoring data from backup
with timestamp: 20230807162819
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for table public.
test_ao from file (table 1 of 1)
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Checking whether segment
agents had errors during restore
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Restoring data from backup
with timestamp: 20230807162904
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for table public.
homes2 from file (table 1 of 4)
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for table public.
test2 from file (table 2 of 4)
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for table public.
homes2a from file (table 3 of 4)
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for table public.
test2a from file (table 4 of 4)
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Checking whether segment
agents had errors during restore
20230807:16:31:57 gprestore:gpadmin:mdw:008603-[INFO]:-Data restore complete
20230807:16:31:57 gprestore:gpadmin:mdw:008603-[INFO]:-Restoring post-data metadata
20230807:16:31:57 gprestore:gpadmin:mdw:008603-[INFO]:-Post-data metadata restore
complete
...
```

The output shows that the restore operation used three backups.

When restoring an from an incremental backup, `gprestore` also lists the backups that are used in the restore operation in the `gprestore` log file.

During the restore operation, `gprestore` displays an error if the full backup or other required incremental backup is not available.

Incremental backup notes

To create an incremental backup, or to restore data from an incremental backup set, you need the complete backup set. When you archive incremental backups, the complete backup set must be archived. You must archive all the files created on the coordinator and all segments.

If you do not specify the `--from-timestamp` option when you create an incremental backup, `gpbackup` uses the most recent backup with a consistent set of options.

If you specify the `--from-timestamp` option when you create an incremental backup, `gpbackup` ensures that the options of the backup that is being created are consistent with the options of the specified backup.

The `gpbackup` option `--with-stats` is not required to be the same for all backups in the backup set. However, to perform a restore operation with the `gprestore` option –with-stats to restore statistics, the backup you specify must have must have used the `--with-stats` when creating the backup.

You can perform a restore operation from any backup in the backup set. However, changes captured in incremental backups later than the backup use to restore database data will not be restored.

When restoring from an incremental backup set, `gprestore` checks the backups and restores each append-optimized table from the most recent version of the append-optimized table in the backup set and restores the heap tables from the latest backup.

The incremental back up set, a full backup and associated incremental backups, must be on a single device. For example, the backups in a backup set must all be on a file system or must all be on a Data Domain system.

If you specify the `gprestore` option `--incremental` to restore data from a specific incremental backup, you must also specify the `--data-only` option. Before performing the restore operation, `gprestore` ensures that the tables being restored exist. If a table does not exist, `gprestore` returns an error and exits.

> **Warning**
>
> Changes to the SynxDB Elastic segment configuration invalidate incremental backups. After you change the segment configuration (add or remove segment instances), you must create a full

backup before you can create an incremental backup.

6.4 High Availability

6.4.1 Time Travel and Flashback Recovery

Time Travel provides SynxDB Elastic with a powerful suite of data protection and historical data analysis capabilities. It allows users to query the state of data at a specific point in the past (that is, historical query) and to quickly recover database objects that were accidentally deleted (that is, flashback recovery).

This feature is a valuable tool for data auditing, recovering from operational errors, and analyzing data evolution trends.

User value

Introducing the Time Travel feature brings the following core values to users:

- **Instant data recovery**: When operational errors occur (such as an `UPDATE` without a `WHERE` clause or a `DELETE` that removes too much data), there's no need to restore from complex backups. You can quickly retrieve the correct data simply by querying its state before the erroneous operation.
- **Historical data auditing and analysis**: Easily query and analyze data at any point in the past for troubleshooting, data auditing, or analyzing trends over time.
- **Simplified database operations**: Quickly recover an accidentally dropped table with a simple `UNDROP` command, significantly reducing operational complexity and recovery time objective (RTO).
- **Flexible query capabilities**: Support joining and analyzing different tables at various historical points in time within a single query.

Typical application scenarios

The following are some typical application scenarios for the Time Travel feature.

Scenario 1: Recover data from operational errors

- **Problem**: A developer forgets to add a `WHERE` clause when executing an `UPDATE` statement in the production environment, causing the entire table's data to be updated incorrectly.
- **Solution**: Use the Time Travel feature to query the table's state before the incorrect operation and restore the correct data, quickly fixing the issue.

- Query the state of table `t1` from 10 minutes ago and table `t2` from 30 minutes ago:

```
SELECT * FROM t1 TRAVELON (now() - interval '10 min'), t2 TRAVELON (now() -
interval '30 min');
```

Scenario 2: Verify historical report data

- **Problem**: This month's sales report data differs significantly from last month's. It is necessary to verify if the source data at that time was accurate.
- **Solution**: Set the state of multiple source tables back to the time when last month's report was generated. Then, perform a join query to compare and identify the cause of the discrepancy.
- Query the state of tables `t1` and `t2` at different specific date and time:

```
SELECT * FROM t1 TRAVELON (timestamp '2024-10-11 10:24:00'), t2 TRAVELON
(timestamp '2024-10-11 11:24:00');
```

Scenario 3: Quickly recover an accidentally dropped table

- **Problem**: While cleaning up test tables, a database administrator accidentally drops an important production table, `t1`.
- **Solution**: Instead of a time-consuming and complex data recovery process, simply execute the `UNDROP` command to restore the table and its data in seconds.
- Restore the dropped table `t1`:

```
UNDROP TABLE t1;
```

How it works

The core of this feature is based on the database's multi-version concurrency control (MVCC) mechanism, which ensures that all historical versions of data are retained in the data tables. The system records a precise commit timestamp for each successfully committed transaction. When a user initiates a historical query, the database uses this sequence of timestamps to compute the data snapshot at the target moment, thus presenting the data state at that time.

To ensure the validity of historical queries, the system must control the cleanup of historical data. For DML operations (like `UPDATE`, `DELETE`), this means managing the progress of `VACUUM`

to prevent premature reclamation of historical data versions. For DDL operations (like `DROP TABLE`), it relies on a recycle bin mechanism to enable the recovery of dropped objects.

Core concepts

Before using Time Travel, it is important to understand the following core concepts.

Data retention period

When data in a table is modified (`UPDATE`, `DELETE`) or the table itself is dropped (`DROP`), SynxDB Elastic does not immediately and permanently remove the data. Instead, it retains these historical data versions or the dropped objects for a period of time, known as the "data retention period".

- **Feature availability**: You can only query historical data or restore dropped objects using Time Travel within the data retention period.
- **Storage costs**: Note that a longer data retention period means more historical data needs to be stored, which will consume additional storage space and increase storage costs. You need to balance your business's recovery time objective (RTO) and cost budget to set a reasonable retention period.

Recycle bin

For DDL operations like `DROP`, information about the dropped object (e.g., a table) is stored in a system-level recycle bin (the `pg_recycle_bin` system table). The `UNDROP` command locates and restores the object by querying information from this recycle bin. The retention time for objects in the recycle bin is also governed by the data retention period.

Configuration and management

To use the Time Travel feature, you need to perform the following configurations.

Enable transaction timestamp tracking

First, you must ensure that the system's `track_commit_timestamp` parameter is enabled. This parameter is used to record the commit timestamp of each transaction and is fundamental to the Time Travel feature. You can set it at the session level or by modifying the configuration file.

```
-- Enable in the current session
SET track_commit_timestamp = on;
```

It is recommended to set `track_commit_timestamp` to `on` in the database configuration file (`postgresql.conf`) to ensure the feature is enabled by default for all sessions.

Set the data retention period

The data retention period determines how long historical data versions and dropped objects are kept before being permanently purged. This period is set globally using the GUC parameter `time_travel_minutes`, with the unit in minutes.

For example, to set the data retention period to 1 hour (60 minutes):

```
SET time_travel_minutes = 60;
```

Similarly, you can persist this session-level setting into the configuration file to make it permanent.

Usage: Query historical data

Time Travel allows you to query historical data using the `TRAVELON` clause in your `SELECT` statement. You can specify different historical points in time for different tables within the same query.

Syntax

```
SELECT ...
FROM table_name [ AS alias ] TRAVELON ( timestamp_expression )
[ ... ]
```

The `TRAVELON` clause follows the table name and takes a timestamp expression as its argument, such as `now() - interval '5 seconds'` or `timestamp '2024-01-01 10:00:00'`.

- Query a data snapshot at a precise point in time.

  ```
  -- Query the state of t1 at a specific time and t2 at another specific time
  SELECT * FROM t1 TRAVELON (timestamp '2024-10-11 10:24:00'), t2 TRAVELON
  (timestamp '2024-10-11 11:24:00');
  ```

- Query data from a period before the current time.

  ```
  -- Query the state of t1 from 1 day ago and t2 from 30 minutes ago
  SELECT * FROM t1 TRAVELON (now() - interval '1 day'), t2 TRAVELON (now() -
  interval '30 min');
  ```

Usage: Restore dropped objects (Flashback)

The flashback feature allows you to quickly recover objects that have been dropped.

Restore a table (`UNDROP TABLE`)

When a table is dropped, you can restore it within the data retention period using the `UNDROP TABLE` command. Before restoring, you must first query the system table `pg_recycle_bin` to find the unique name of the table in the recycle bin.

Syntax:

```
UNDROP TABLE <new_table_name> AS <recycle_bin_table_name>;
```

- `<new_table_name>`: The new name for the restored table.
- `<recycle_bin_table_name>`: The name of the table to be restored, as it exists in `pg_recycle_bin`.

Example:

```
-- Assume table t1 was accidentally dropped.
DROP TABLE t1;

-- 1. Query the recycle bin for the unique name of the dropped table.
SELECT relname FROM pg_recycle_bin WHERE orig_relname = 't1';

-- Assume the query result is 't1_1678886400'.

-- 2. Use the retrieved name to restore the table and name it t1.
UNDROP TABLE t1 AS "t1_1678886400";
```

Limitations and considerations

- **Supported objects**: Time Travel and flashback generally support permanent tables.
- **Unsupported objects**: Non-persistent objects such as temporary tables and external tables do not typically support Time Travel.
- **Data retention period is key**: All Time Travel operations are strictly limited by the configured data retention period. Once this period is exceeded, historical data may be permanently deleted.

- **Impact of `VACUUM`**: The background `VACUUM` process is responsible for cleaning up expired data versions. The data retention period setting directly affects `VACUUM`'s behavior.
- **Storage costs**: Enabling Time Travel and setting a long retention period will increase your storage costs. Be sure to monitor and plan accordingly.

6.4.2 Cluster Failover

SynxDB Elastic v4 features fast failover. When a data warehouse node or other service in the cluster fails, the cluster can quickly detect the failure and automatically recover the failed node and its stored data, ensuring a rapid restoration to the state before the failure.

Failover mechanism

SynxDB Elastic is deployed in a containerized environment based on Kubernetes (K8s). Components and services that form the database cluster are deployed in different pods and are managed and scheduled by the K8s cluster.

When a data warehouse node (Segment), data warehouse proxy service, or cluster-level control service in a SynxDB Elastic cluster fails, K8s can automatically restart the corresponding Pod and re-associate its Persistent Volume in a short time. Data from before the failure is completely preserved and restored. The restarted Pod has the same network configuration as before the failure. Once the failed Pod restarts and enters the `Running` state, the database can resume normal operation.

Note: Before the failed Pod enters the `Running` state, the database cluster or the data warehouse corresponding to the Pod will not function normally.

Failover scenarios

This section lists the following failover scenarios and provides corresponding examples.

- Failover for cluster metadata proxy service (example Pod: `cloudberry-proxy-0`)
- Failover for cluster metadata `safekeeper` service (example Pod: `safekeeper-0`)
- Failover for cluster metadata `pageserver` service (example Pod: `pageserver-0`)
- Failover for data warehouse proxy service in the cluster (example Pod: `cloudberry-proxy-0`)
- Failover for data warehouse node in the cluster (example Pod: `wh9527-0`)

The following examples simulate a failure by deleting the Pod of the corresponding service in the K8s cluster. After deleting the Pod, the success of the failover is determined by observing the Pod restart process and testing the functionality of the corresponding module in the database.

Prerequisites

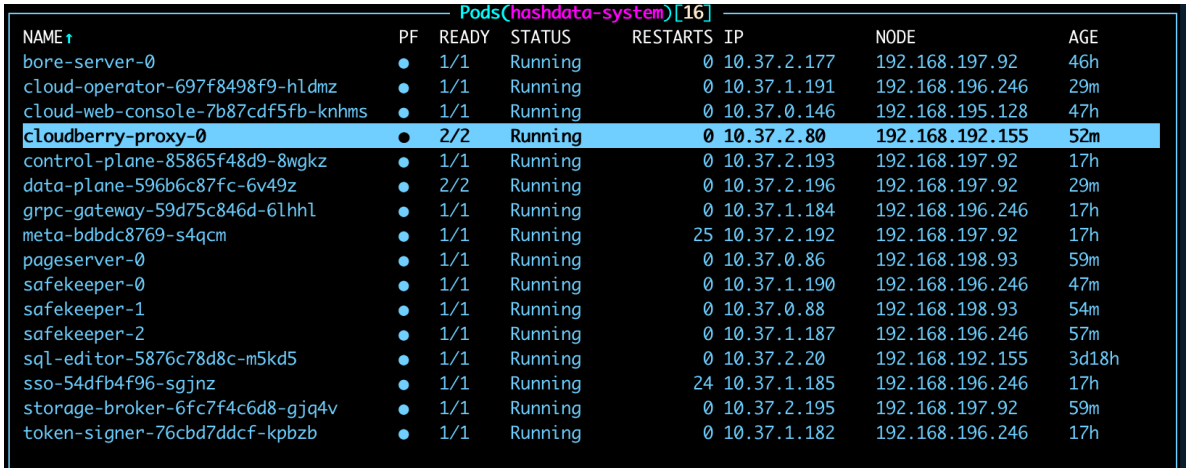
Before proceeding with the following tests, ensure the following prerequisites are met:

1. An example SynxDB Elastic cluster has been created and is accessible.
2. The Kubernetes command-line tool `kubectl` and the `k9s` console are installed locally.

Cluster metadata proxy service failover

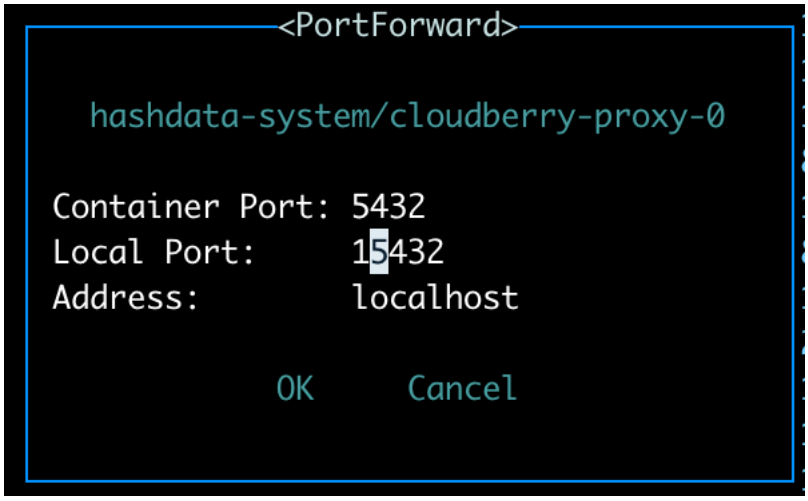
This example simulates the failover of the SynxDB Elastic cluster metadata proxy service after a failure.

1. Access the `k9s` console and locate the `cloudberry-proxy-0` Pod in the `synxdb-system` namespace.



2. Set a local forwarding port for `cloudberry-proxy-0`. Use `shift + f` to open the port forwarding settings, enter the container port (default is `5432`) and the local port to forward to. The address can be the default `localhost`.

   Note: Avoid setting the local port to `5432` to prevent conflicts with a local PostgreSQL installation.

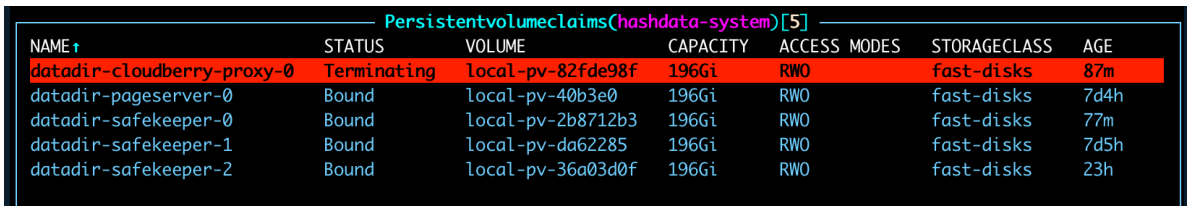


3. Log in to the cluster using the forwarding port and address set in the previous step: `psql -h localhost -p 15432 -U gpadmin -d postgres`.

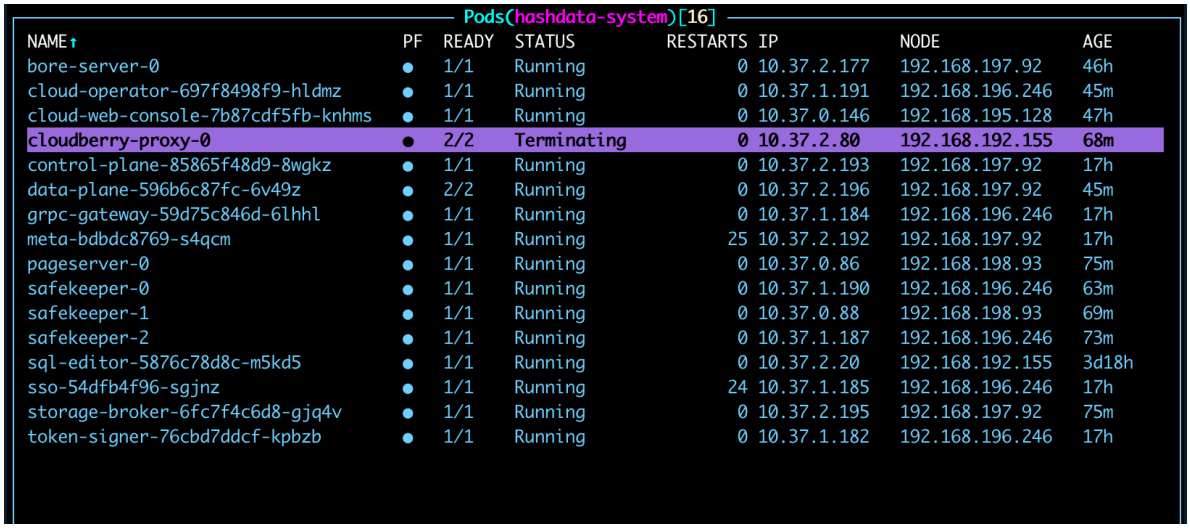
4. Enter the `cloudmeta` database and verify that data can be accessed:

```
--Enter the cloudmeta database
\c cloudmeta
--List tables in the current database
\dt
```

5. In the `k9s` cluster, switch to the Persistent Volume Claim (PVC) view: `:pvc`. Find the corresponding storage claim and delete it using `CTRL+D`.



6. In the Pod list, delete `cloudberry-proxy-0`.



After the Pod is deleted, k8s will restart it. The metadata cluster will be temporarily inaccessible until the Pod's status returns to `Running`.

7. Wait for the Pod to restart. Once its status is `Running`, set up local port forwarding again and access the `cloudmeta` database. The database should now be accessible.

```
cloudmeta=# \dt
                 List of relations
 Schema |            Name             | Type  |  Owner
--------+-----------------------------+-------+---------
 public | __diesel_schema_migrations | table | gpadmin
 public | accounts                    | table | gpadmin
 public | clouds                      | table | gpadmin
 public | clusters                    | table | gpadmin
 public | config_files                | table | gpadmin
 public | domains                     | table | gpadmin
 public | organizations               | table | gpadmin
 public | privileges                  | table | gpadmin
 public | regions                     | table | gpadmin
 public | users                       | table | gpadmin
 public | users_3rd                   | table | gpadmin
 public | users_privileges            | table | gpadmin
(12 rows)
```

Cluster metadata `safekeeper` service failover

This example simulates the failover of the SynxDB Elastic cluster metadata `safekeeper-0` service after a failure. For specific screenshots and commands, refer to the cluster metadata proxy service failover example.

1. Access the `k9s` console and locate the `safekeeper-0` Pod in the `synxdb-system` namespace.
2. Switch to the PVC view, find the corresponding storage claim, and delete it.
3. Delete the `safekeeper-0` Pod. k8s will automatically restart it. The database remains accessible during this period.
4. The `safekeeper-0` Pod’ s status returns to `Running`, and the failover is complete.

Cluster metadata `pageserver` service failover

This example simulates the failover of the SynxDB Elastic cluster metadata `pageserver` service after a failure. For specific screenshots and commands, refer to the cluster metadata proxy service failover example.

1. Access the `k9s` console and locate the `pageserver-0` Pod in the `synxdb-system` namespace.
2. Switch to the PVC view, find the corresponding storage claim, and delete it.
3. Delete the `pageserver-0` Pod. k8s will automatically restart it. The database remains accessible during this period.
4. The `pageserver-0` Pod’ s status returns to `Running`, and the failover is complete.

Data warehouse proxy service failover

This example simulates the failover of the SynxDB Elastic data warehouse cluster proxy service after a failure.

1. In `k9s`, go to the data warehouse namespace (e.g., `cloud007-513d75d5-3033-4649-b913-07f6115ec41a+`) and locate the proxy service Pod `cloudberry-proxy-0`.
2. Set a local forwarding port for `cloudberry-proxy-0`. Use `shift + f` to open the port

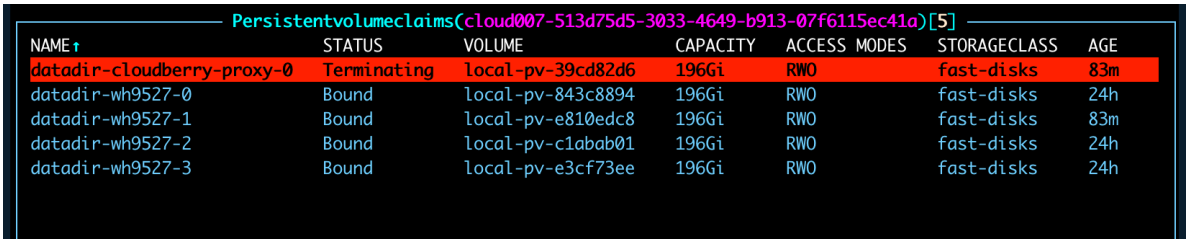
forwarding settings, enter the container port (default is `5432`) and the local port to forward to. The address can be the default `localhost`.

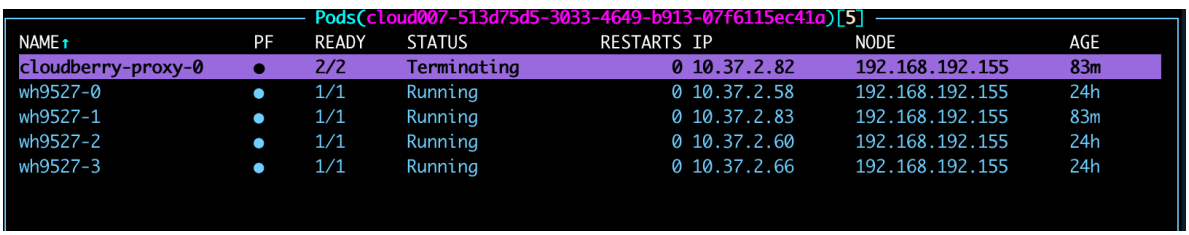
3. Log in to the cluster locally using the forwarding port and address set in the previous step:
   `psql -h localhost -p 15432 -U gpadmin -d postgres`

4. Create a test database and insert test data:

```
--Creates the database.
CREATE DATABASE test_db;

--Enters the database.
\c test_db;

--Creates a table and inserts data.
CREATE TABLE projects (
    project_id SERIAL PRIMARY KEY,
    project_name TEXT,
    project_manager TEXT,
    department TEXT
) USING heap;

INSERT INTO projects (project_name, project_manager, department) VALUES
('Project Alpha', 'Alice', 'Engineering'),
('Project Beta', 'Bob', 'HR'),
('Project Gamma', 'Charlie', 'Sales');
```

5. Delete the `cloudberry-proxy-0` Pod and its corresponding PVC. At this point, `test_db` will be temporarily inaccessible.





6. When the Pod's status automatically changes to `Running`, `test_db` becomes accessible again.

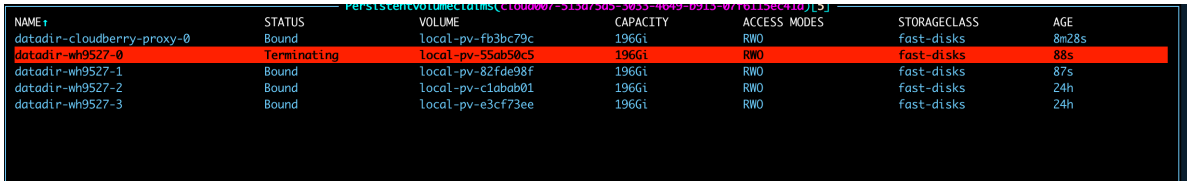
```
test_db=# select * from projects;
 project_id | project_name  | project_manager | department
------------+---------------+-----------------+-------------
          1 | Project Alpha | alice           | Engineering
          2 | Project Beta  | bob             | HR
          3 | Project Gamma | charlie         | Sales
(3 rows)
```

Data warehouse node failover

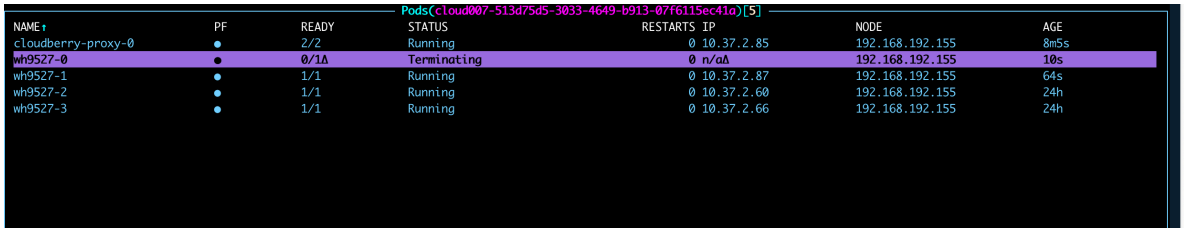
This example simulates the failover of a SynxDB Elastic cluster data node after a failure.

1. In `k9s`, go to the data warehouse namespace (e.g., `cloud007-513d75d5-3033-4649-b913-07f6115ec41a+`) and locate the data node Pod (e.g., `wh9527-0`).

2. Go to the PVC list, locate the corresponding PVC, and delete it.



3. Delete the Pod `wh9527-0`. At this point, `test_db` will be briefly inaccessible.



4. The deleted Pod is automatically restarted. When its status changes to `Running`, database access is restored.

Chapter 7

Configure Security and Permissions

7.1 Set Password Profile

Profile refers to the password policy configuration, which is used to control the password security policy of users in SynxDB Elastic. You can bind a profile to one or more users to control the password security policy of database users. Profile defines the rules for user management and password reuse. With Profile, the database administrator can use SQL to force some constraints, such as locking accounts after login failures or controlling the number of password reuses.

Note

- In general, Profile includes password policy and user resource usage restrictions. Profile in SynxDB Elastic only supports password policy. "Profile" mentioned in this document refers to password policy configuration.
- Only superusers can create or modify Profile policies, and superusers are not restricted by any Profile policies. Profile policies will take effect only when regular users are allowed to use Profile.

The Profile feature is enabled by default. The default value of `enable_password_profile` is `true`.

7.1.1 Implementation principle

Similar to the Autovacuum mechanism, Profile introduces the Login Monitor Launcher and Login Monitor Worker processes. When user login verification fails, SynxDB Elastic will send a signal to the postmaster. After receiving the signal, the postmaster will send a signal to the launcher process. After receiving the signal, the launcher process will notify the postmaster to launch a worker process to perform the metadata write-back operation, and notify the user process and the launcher process after completion.

7.1.2 Set password policies using SQL

Database administrators can use SQL to set Profile. The following parameters are commonly used.

Parameter	Description
`FAILED_LOGIN_ATTEMPTS`	• The maximum number of failed logins before the user account is locked. • Valid values include `-2` (unlimited), `-1` (default), and `1` to `9999`. `0` is an invalid value.
`PASSWORD_LOCK_TIME`	• The lock time (in hours) after multiple consecutive failed login attempts. • Valid values are `-2` to `9999`. `0` is a valid value.
`PASSWORD_REUSE_MAX`	• The number of historical password reuses. • Valid values are `-2` to `9999`. `0` is a valid value.

CREATE PROFILE

Creates a profile and sets its password policy.

```
CREATE PROFILE profile LIMIT
  password_parameters ... ;

password_parameters:
{ { FAILED_LOGIN_ATTEMPTS
  | PASSWORD_REUSE_MAX
  | PASSWORD_LOCK_TIME
  }
  expr
}
```

ALTER PROFILE

Modifies a password policy.

```
ALTER PROFILE profile LIMIT
   password_parameters ... ;
```

DROP PROFILE

Deletes a profile.

```
DROP PROFILE profile;
```

CREATE USER ···PROFILE

Creates a user and sets its profile.

```
CREATE USER user PROFILE profile;
```

ALTER USER ···PROFILE

Modifies the profile of `user`.

```
ALTER USER user PROFILE profile;
```

CREATE USER ···ENABLE/DISABLE PROFILE

Creates a user and specifies their permission to use Profile. `ENABLE PROFILE` grants permission, while `DISABLE PROFILE` denies it. By default, newly created users are not permitted to use Profile.

```
CREATE USER user
  { ENABLE | DISABLE }
  PROFILE;
```

ALTER USER ···ENABLE/DISABLE PROFILE

Sets whether a user is allowed to use Profile.

```
ALTER USER user
  { ENABLE | DISABLE }
  PROFILE;
```

CREATE USER ···ACCOUNT LOCK/UNLOCK

Creates a user and sets whether the user is locked. `ACCOUNT LOCK` means that the user is locked and cannot log in. `ACCOUNT UNLOCK` means that the user is not locked and can log in. The default created user is not locked.

```
CREATE USER user ACCOUNT
  { LOCK | UNLOCK };
```

ALTER USER ···ACCOUNT LOCK/UNLOCK

Sets whether to lock the account of a user.

```
ALTER USER user ACCOUNT
  { LOCK | UNLOCK };
```

7.1.3 Check password policy information in system tables

After applying the password configuration policy, SynxDB Elastic will update metadata: add two system tables `pg_profile` and `pg_password_history`, and add some fields to the system tables/views `pg_authid` and `pg_roles`. For example:

- **pg_catalog.pg_roles**: In `pg_roles`, the `rolprofile`, `rolaccountstatus`, and `rolfailedlogins` fields are added to record database users who use Profile, the account status, and the number of failed logins.

```
View "pg_catalog.pg_roles"
    Column       |           Type           | Collation | Nullable | Default |
Storage  | Description
-----------------+--------------------------+-----------+----------+---------+--
--------+-------------
rolname          | name                     |           |          |         |
plain    |
rolsuper         | boolean                  |           |          |         |
plain    |
```

(continues on next page)

(continued from previous page)

```
rolinherit        | boolean                  |           |          |          |
plain    |
rolcreaterole     | boolean                  |           |          |          |
plain    |
rolcreatedb       | boolean                  |           |          |          |
plain    |
rolcanlogin       | boolean                  |           |          |          |
plain    |
rolreplication    | boolean                  |           |          |          |
plain    |
rolconnlimit      | integer                  |           |          |          |
plain    |
rolprofile        | name                     |           |          |          |
plain    |
rolaccountstatus  | smallint                 |           |          |          |
plain    |
rolfailedlogins   | integer                  |           |          |          |
plain    |
rolpassword       | text                     |           |          |          |
extended |
rolvaliduntil     | timestamp with time zone |           |          |          |
plain    |
rolbypassrls      | boolean                  |           |          |          |
plain    |
rolconfig         | text[]                   | C         |          |          |
extended |
rolresqueue       | oid                      |           |          |          |
plain    |
oid               | oid                      |           |          |          |
plain    |
rolcreaterextgpfd | boolean                  |           |          |          |
plain    |
rolcreaterexthttp | boolean                  |           |          |          |
plain    |
rolcreatewextgpfd | boolean                  |           |          |          |
plain    |
rolresgroup       | oid                      |           |          |          |
plain    |
```

- **pg_catalog.pg_authid**: In `pg_authid`, the `rolprofile`, `rolaccountstatus`,

`rolfailedlogins`, `rolpasswordsetat`, `rollockdata`, and `rolpasswordexpire` are added to record the database users who use Profile, the account status, the number of failed logins, password setting time, account lock time, and password expiration time.

```
Table "pg_catalog.pg_authid"
    Column       |           Type           | Collation | Nullable | Default |
Storage  | Compression | Stats target | Description
-----------------+--------------------------+-----------+----------+---------+--
-------+-------------+--------------+-------------
oid               | oid                      |           | not null |         |
plain    |             |              |
rolname           | name                     |           | not null |         |
plain    |             |              |
rolsuper          | boolean                  |           | not null |         |
plain    |             |              |
rolinherit        | boolean                  |           | not null |         |
plain    |             |              |
rolcreaterole     | boolean                  |           | not null |         |
plain    |             |              |
rolcreatedb       | boolean                  |           | not null |         |
plain    |             |              |
rolcanlogin       | boolean                  |           | not null |         |
plain    |             |              |
rolreplication    | boolean                  |           | not null |         |
plain    |             |              |
rolbypassrls      | boolean                  |           | not null |         |
plain    |             |              |
rolconnlimit      | integer                  |           | not null |         |
plain    |             |              |
rolenableprofile  | boolean                  |           | not null |         |
plain    |             |              |
rolpassword       | text                     | C         |          |         |
extended |             |              |
rolvalidunti      | timestamp with time zone |           |          |         |
plain    |             |              |
rolprofile        | oid                      |           | not null |         |
plain    |             |              |
rolaccountstatus  | smallint                 |           | not null |         |
plain    |             |              |
rolfailedlogins   | integer                  |           | not null |         |
plain    |             |              |
```

(continues on next page)

(continued from previous page)

```
rolpasswordsetat   | timestamp with time zone |           |          |         |
plain    |              |              |
rollockdate        | timestamp with time zone |           |          |         |
plain    |              |              |
rolpasswordexpire  | timestamp with time zone |           |          |         |
plain    |              |              |
rolresqueue        | oid                      |           |          |         |
plain    |              |              |
rolcreaterextgpfd  | boolean                  |           |          |         |
plain    |              |              |
rolcreaterexthttp  | boolean                  |           |          |         |
plain    |              |              |
rolcreatewextgpfd  | boolean                  |           |          |         |
plain    |              |              |
rolresgroup        | oid                      |           |          |         |
plain    |              |              |
Tablespace: "pg_global"
Access method: heap
```

- **pg_catalog.pg_profile**

  The newly added `pg_profile` system table is as follows:

```
Table "pg_catalog.pg_profile"
        Column           |  Type   | Collation | Nullable | Default | Storage |
Compression | Stats target | Description
-------------------------+---------+-----------+----------+---------+---------+---
----------+--------------+-------------
oid                      | oid     |           | not null |         | plain   |
        |              |
prfname                  | name    |           | not null |         | plain   |
        |              |
prffailedloginattempts   | integer |           | not null |         | plain   |
        |              |
prfpasswordlocktime      | integer |           | not null |         | plain   |
        |              |
prfpasswordlifetime      | integer |           | not null |         | plain   |
        |              |
prfpasswordgracetime     | integer |           | not null |         | plain   |
        |              |
```

(continues on next page)

(continued from previous page)

```
prfpasswordreusetime     | integer |           | not null |         | plain   |
        |              |
prfpasswordreusemax      | integer |           | not null |         | plain   |
        |              |
prfpasswordallowhashed   | integer |           | not null |         | plain   |
        |              |
prfpasswordverifyfuncdb | oid      |           |          |         | plain   |
        |              |
prfpasswordverifyfunc    | oid      |           |          |         | plain   |
        |              |
Tablespace: "pg_global"
Access method: heap
```

> **Note**
>
> The fields in the `pg_profile` table are described as follows:
>
> - `oid`: used to uniquely identify each profile record.
> - `prfname`: the name of the configuration file.
> - `prffailedloginattempts`: the number of failed login attempts allowed before an account is locked.
> - `prfpasswordlocktime`: the password lock time. If an account is locked due to failed login attempts, this field defines how long the lock lasts.
> - `prfpasswordreusemax`: the number of new passwords that must be set before the old password can be reused.
> - Other fields in the table are not valid.

- **pg_catalog.pg_password_history**

```
Table "pg_catalog.pg_password_history"
        Column          |           Type           | Collation | Nullable | Default
| Storage  | Compression | Stats target | Description
----------------------+--------------------------+-----------+----------+--------
-+----------+-------------+--------------+-------------
```

(continues on next page)

(continued from previous page)

```
passhistroleid        | oid                         |           | not null |
| plain    |              |              |
passhistpasswordsetat | timestamp with time zone |           | not null |
| plain    |              |              |
passhistpassword      | text                        | C         | not null |
| extended |              |              |
Tablespace: "pg_global"
Access method: heap
```

> **Note**
>
> The fields in the `pg_password_history` table are described as follows:
>
> - `passhistroleid`: a unique ID that identifies the user or role related to this password history.
> - `passhistpasswordsetat`: a timestamp field with the time zone that records the exact time when the password was set or last modified.
> - `Passhistpassword`: the ciphertext that stores historical passwords.

7.1.4 Default password policy

When you create a user, SynxDB Elastic applies the default Profile to the user by default if no specific password policy is specified. The default Profile is the default password policy during system initialization. The default Profile in SynxDB Elastic is the `pg_default` row in the `pg_profile` table. The `pg_default` row defines default values for the Profile parameters, and only superusers can modify these parameters.

If a user sets a parameter with the default value `-1`, the parameter will get its value from `pg_default`. The default values of `pg_default` are as follows. Refer to *Scenario 3* for how to use the default Profile.

```
\x
Expanded display is on.

-- Checks the values of the default Profile in pg_profile
SELECT * FROM pg_profile WHERE prfname = 'pg_default';
```

(continues on next page)

(continued from previous page)

```
-[ RECORD 1 ]-----------+-----------
oid                     | 10140
prfname                 | pg_default
prffailedloginattempts  | -2
prfpasswordlocktime     | -2
prfpasswordlifetime     | -2
prfpasswordgracetime    | -2
prfpasswordreusetime    | -2
prfpasswordreusemax     | -2
prfpasswordallowhashed  | 1
prfpasswordverifyfuncdb |
prfpasswordverifyfunc   |
```

> **Note**
>
> The `pg_default` row cannot be renamed or dropped by any user (including superusers).

7.1.5 Scenario examples

This section introduces typical usage scenarios of Profile.

Create a password policy

Before creating a password policy, you need to create a profile first and bind a database user to the profile. For example:

```
-- Creates a profile and a user.
CREATE PROFILE myprofile;
CREATE USER myuser PASSWORD 'mypassword';

-- Binds the profile to the user.
ALTER USER myuser PROFILE myprofile;

-- Checks the relationship between the user and the profile.
SELECT rolname, rolprofile FROM pg_roles WHERE rolname = 'myuser';
 rolname | rolprofile
---------+------------
 myuser  | myprofile
(1 row)
```

Scenario 1: Set the maximum number of failed login attempts and password lock time

Modify a Profile to set the maximum number of failed login attempts to 3 and the password lock time to 2 hours.

> **Note**
>
> When multiple users fail to log in, the result return speed may slow down.

```
ALTER PROFILE myprofile LIMIT
  FAILED_LOGIN_ATTEMPTS 3
  PASSWORD_LOCK_TIME 2;

-- Allows the user myuser to use Profile.
ALTER USER myuser ENABLE PROFILE;



-- Checks the details in the catalog table (the pg_profile is the catalog table that
stores all the details related to the user profile).
-- Note that the time shown below is in seconds.

SELECT prfname, prffailedloginattempts, prfpasswordlocktime
FROM pg_profile
WHERE prfname = 'myprofile';
  prfname   | prffailedloginattempts | prfpasswordlocktime
-----------+------------------------+---------------------
 myprofile |                      3 |                   2
(1 row)

SELECT rolname, rolprofile, get_role_status('myuser'), rolfailedlogins, rollockdate
FROM pg_roles
WHERE rolname = 'myuser';
 rolname | rolprofile | get_role_status | rolfailedlogins | rollockdate
---------+------------+-----------------+-----------------+-------------
 myuser  | myprofile  | OPEN            |               0 |
(1 row)
```

The current user `myuser` does not have any login failures yet. The `pg_roles` system table shows that the user's status is `OPEN`, and the `rolfailedlogins` field in the system table `pg_roles` is `0`. Now `myuser` attempts a failed login and queries `pg_roles` again.

```
\c - myuser
Password for user myuser:
FATAL:  password authentication failed for user "myuser"
Previous connection kept

SELECT rolname, rolprofile, get_role_status('myuser'), rolfailedlogins, rollockdate
FROM pg_roles
WHERE rolname = 'myuser';
 rolname | rolprofile | get_role_status | rolfailedlogins | rollockdate
---------+------------+-----------------+-----------------+-------------
 myuser  | myprofile  | OPEN            |               1 |
(1 row)
```

The above result shows that the user status is still `OPEN`, and `rolfailedlogins` have increased to `1`. If the account fails to log in again, the `rolfailedlogins` will continue to increase until the account is locked:

```
\c - myuser
Password for user myuser:
FATAL:  role "myuser" is locked
Previous connection kept

SELECT rolname, rolprofile, get_role_status('myuser'), rolfailedlogins, rollockdate
FROM pg_roles
WHERE rolname = 'myuser';
 rolname | rolprofile | get_role_status | rolfailedlogins |           rollockdate
---------+------------+-----------------+-----------------+---------------------------
-------
 myuser  | myprofile  | LOCKED(TIMED)   |               4 |13-MAR-23 12:25:50.811022
+08:00
(1 row)
```

The user account is locked because of too many failed logins. The user status has changed to `LOCKED(TIMED)` and the user account will be automatically unlocked after 2 hours (controlled by the `PASSWORD_LOCK_TIME` parameter).

At the same time, the system records the timestamp of the user account lock. When the lock period expires, the user account's status will change back to `OPEN` and login will be allowed. If the user fails to log in several times and the number of failures does not exceed the limit of `FAILED_LOGIN_ATTEMPTS`, then the user can log in successfully, and the system will reset

`rolfailedlogins` to `0`. See the following example:

```
\c - myuser
Password for user myuser:
You are now connected to database "postgres" as user "myuser".

SELECT rolname, rolprofile, get_role_status('myuser'), rolfailedlogins, rollockdate
FROM pg_roles
WHERE rolname = 'myuser';
 rolname | rolprofile | get_role_status | rolfailedlogins |           rollockdate
---------+------------+-----------------+-----------------+-------------------------------
-------
 myuser  | myprofile  | OPEN            |               0 |13-MAR-23 12:25:50.811022
+08:00
(1 row)
```

> **Note**
>
> If `PASSWORD_LOCK_TIME` is manually set to `0`, the user account will never be locked.

Scenario 2: Set the number of historical password reuses

With the `PASSWORD_REUSE_MAX` parameter, you can prevent users from setting recently used passwords. If you want to prevent users from using the last two historical passwords, you can use `ALTER PROFILE` to modify this parameter. For example:

```
ALTER PROFILE myprofile LIMIT
  PASSWORD_REUSE_MAX 2;

-- Checks the Catalog table and finds that the number of permitted historical password
reuses is 2.
  SELECT prfname, prfpasswordreusemax
  FROM pg_profile
  WHERE prfname = 'myprofile';
    prfname   | prfpasswordreusemax
-----------+---------------------
 myprofile |                   2
(1 row)
```

```
ALTER USER myuser PASSWORD 'mynewpassword';

ALTER USER myuser PASSWORD 'mypassword';
ERROR:  The new password should not be the same with latest 2 history password
```

The above result shows that `mypassword` is not allowed to be reused as a new password because it has been used before. To set a new password, make sure that two different in-between passwords have been set before reusing an old password.

```
ALTER USER myuser PASSWORD 'mypassword2'; -- Second password change
ALTER USER myuser PASSWORD 'mypassword';
```

> **Note**
>
> If `PASSWORD_REUSE_MAX` is set to `0`, the password can never be changed. If set to `-2`(`UNLIMITED`), the history password can only be used after `9999` passwords have been set.

Scenario 3: Use DEFAULT PROFILE

If you do not explicitly specify a parameter value when creating a profile, the parameter value in the `pg_profile` table is `-1` by default, which means that SynxDB Elastic will obtain the value of this parameter from `pg_default`.

Take `FAILED_LOGIN_ATTEMPTS` as an example:

```
--Creates a profile without explicitly specifying any parameter values.
CREATE PROFILE myprf;

-- Checks the default parameter value of myprf in pg_profile.

SELECT * FROM pg_profile WHERE prfname = 'myprf';
-[ RECORD 1 ]-----------+------
oid                     | 16386
prfname                 | myprf
prffailedloginattempts  | -1
prfpasswordlocktime     | -1
prfpasswordlifetime     | -1
prfpasswordgracetime    | -1
```

(continues on next page)

(continued from previous page)

```
prfpasswordreusetime    | -1
prfpasswordreusemax     | -1
prfpasswordallowhashed  | -1
prfpasswordverifyfuncdb |
prfpasswordverifyfunc   |
```

The above example creates a profile named `myprf`. Because no parameter values are set explicitly, all parameter values are `-1` by default, which means that any user bound to this profile will use the parameter values in `pg_default`.

The following example sets the `FAILED_LOGIN_ATTEMPTS` parameter of `pg_default` to `1` and creates a test.

```
-- Sets the default value of FAILED_LOGIN_ATTEMPTS in pg_default to 1.
-- All users who do not specify a profile will have their accounts locked after a
failed login.

ALTER PROFILE pg_default LIMIT FAILED_LOGIN_ATTEMPTS 1;
\x
Expanded display is on.

-- Checks the default value of pg_default in pg_profile.
SELECT * FROM pg_profile WHERE prfname = 'pg_default';
-[ RECORD 1 ]-----------+-----------
oid                     | 10140
prfname                 | pg_default
prffailedloginattempts  | 1
prfpasswordlocktime     | -2
prfpasswordlifetime     | -2
prfpasswordgracetime    | -2
prfpasswordreusetime    | -2
prfpasswordreusemax     | -2
prfpasswordallowhashed  | 1
prfpasswordverifyfuncdb |
prfpasswordverifyfunc   |

CREATE USER mynewuser PASSWORD 'mynewpassword' ENABLE PROFILE;

SELECT rolname, rolprofile, get_role_status('mynewuser'), rolfailedlogins, rollockdate
```

(continues on next page)

(continued from previous page)

```
FROM pg_roles
WHERE rolname = 'mynewuser';
  rolname   | rolprofile | get_role_status | rolfailedlogins | rollockdate
------------+------------+-----------------+-----------------+-------------
 mynewuser | pg_default | OPEN            |               0 |
(1 row)
```

From the above result, you can see that the user's profile is `pg_default` and the user account status is `OPEN`. Next, try logging into the account using a wrong password.

```
\c - mynewuser
Password for user mynewuser:
FATAL:  password authentication failed for user "mynewuser"
Previous connection kept

SELECT rolname, rolprofile, get_role_status('mynewuser'), rolfailedlogins, rollockdate
FROM pg_roles
WHERE rolname = 'mynewuser';
  rolname   | rolprofile | get_role_status | rolfailedlogins |           rollockdate
------------+------------+-----------------+-----------------+----------------------
-------------
 mynewuser | pg_default | LOCKED(TIMED)   |               1 | 12-MAR-23 09:36:42.
132231 +08:00
(1 row)
```

Because `FAILED_LOGIN_ATTEMPTS` of `pg_default` is `1`, the account is locked after one failed login.

Next, set the user's profile to `myprf`, and then test the same operation to observe the result. The user needs to be unlocked before the test.

```
ALTER USER mynewuser ACCOUNT unlock PROFILE myprf;

SELECT rolname, rolprofile, get_role_status('mynewuser'), rolfailedlogins, rollockdate
FROM pg_roles
WHERE rolname = 'mynewuser';
  rolname   | rolprofile | get_role_status | rolfailedlogins |           rollockdate
------------+------------+-----------------+-----------------+----------------------
-------------
```

(continues on next page)

(continued from previous page)

```
 mynewuser | myprf      | OPEN             |                0 | 12-MAR-23 09:36:42.
132231 +08:00
(1 row)
```

The result shows that the user status changes back to `OPEN` and the profile changes to `myprf`. `rollockdate` still exists as expected. Next, use a wrong password to log in again and observe the returned result.

```
\c - mynewuser
Password for user mynewuser:
FATAL:  password authentication failed for user "mynewuser"
Previous connection kept

SELECT rolname, rolprofile, get_role_status('mynewuser'), rolfailedlogins, rollockdate
FROM pg_roles
WHERE rolname = 'mynewuser';
  rolname  | rolprofile | get_role_status | rolfailedlogins |           rollockdate
-----------+------------+-----------------+-----------------+-----------------------
-------------
 mynewuser | myprf      | LOCKED(TIMED)   |               1 | 12-MAR-23 09:38:26.
753832 +08:00
(1 row)
```

As expected, the user account status is locked again.

Scenario 4: Superuser locks or unlocks user account

A superuser can lock or unlock a user account. For example:

```
ALTER USER myuser ACCOUNT LOCK;

SELECT rolname, rolprofile, get_role_status('myuser'), rolfailedlogins, rollockdate
FROM pg_roles
WHERE rolname = 'myuser';
 rolname | rolprofile | get_role_status | rolfailedlogins |           rollockdate
---------+------------+-----------------+-----------------+-----------------------
-----------
 myuser  | myprofile  | LOCKED          |               0 | 13-MAR-23 12:25:50.811022
+08:00
(1 row)
```

(continues on next page)

(continued from previous page)

```
ALTER USER myuser ACCOUNT UNLOCK;

SELECT rolname, rolprofile, get_role_status('myuser'), rolfailedlogins, rollockdate
FROM pg_roles
WHERE rolname = 'myuser';
 rolname | rolprofile | get_role_status | rolfailedlogins |           rollockdate
---------+------------+-----------------+-----------------+-----------------------
-----------
 myuser  | myprofile  | OPEN            |               0 | 13-MAR-23 13:23:50.83928
+08:00
(1 row)
```

Scenario 5: Enable Profile for regular users

By default, a newly created regular user is not bound to a Profile. To use Profile, you need to specify it explicitly. See the following example:

```
CREATE USER myuser1;

SELECT rolname, rolprofile, get_role_status('myuser1'), rolfailedlogins, rollockdate,
rolenableprofile
FROM pg_roles
WHERE rolname like 'myuser1';
 rolname | rolprofile | get_role_status | rolfailedlogins | rollockdate |
rolenableprofile
---------+------------+-----------------+-----------------+-------------+-------------
-----
 myuser1 | pg_default | OPEN            |               0 |             | f
(1 row)

CREATE USER myuser2 ENABLE PROFILE;
SELECT rolname, rolprofile, get_role_status('myuser2'), rolfailedlogins, rollockdate,
rolenableprofile
FROM pg_roles
WHERE rolname = 'myuser2';
 rolname | rolprofile | get_role_status | rolfailedlogins | rollockdate |
rolenableprofile
---------+------------+-----------------+-----------------+-------------+-------------
-----
```

(continues on next page)

(continued from previous page)

```
 myuser2 | pg_default | OPEN            |               0 |             | t
(1 row)

CREATE USER myuser3 DISABLE PROFILE;
SELECT rolname, rolprofile, get_role_status('myuser3'), rolfailedlogins, rollockdate,
rolenableprofile
FROM pg_roles
WHERE rolname = 'myuser3';
 rolname | rolprofile | get_role_status | rolfailedlogins | rollockdate |
rolenableprofile
---------+------------+-----------------+-----------------+-------------+-------------
-----
 myuser3 | pg_default | OPEN            |               0 |             | f
(1 row)
```

After creating a user, you can modify whether to use Profile through the SQL statement `ALTER USER ENABLE/DISABLE PROFILE`.

```
ALTER USER myuser1 ENABLE PROFILE;

SELECT rolname, rolprofile, get_role_status('myuser1'), rolfailedlogins, rollockdate,
rolenableprofile
FROM pg_roles
WHERE rolname = 'myuser1';
 rolname | rolprofile | get_role_status | rolfailedlogins | rollockdate |
rolenableprofile
---------+------------+-----------------+-----------------+-------------+-------------
-----
 myuser1 | pg_default | OPEN            |               0 |             | t
(1 row)

ALTER USER myuser1 DISABLE PROFILE;
SELECT rolname, rolprofile, get_role_status('myuser1'), rolfailedlogins, rollockdate,
rolenableprofile
FROM pg_roles
WHERE rolname = 'myuser1';
 rolname | rolprofile | get_role_status | rolfailedlogins | rollockdate |
rolenableprofile
---------+------------+-----------------+-----------------+-------------+-------------
-----
```

(continues on next page)

(continued from previous page)

```
 myuser1 | pg_default | OPEN             |               0 |              | f
(1 row)
```

7.2 Use pgcrypto for Data Encryption

`pgcrypto` is an encryption module for SynxDB Elastic that provides cryptographic functions.

This module is "trusted," meaning that non-superusers with `CREATE` privileges in the current database can install the `pgcrypto` module.

```
CREATE EXTENSION pgcrypto;
```

7.2.1 Notes on OpenSSL-related algorithm support

When building `pgcrypto`, you can optionally include OpenSSL (minimum version 1.0.1) using the `--with-openssl` option. The cryptographic algorithms supported by SynxDB Elastic differ depending on whether OpenSSL is included.

For example, `pgcrypto` built with the `--with-openssl` option supports DES/3DES/CAST5 algorithms but does not support SM3/SM4 algorithms. Conversely, `pgcrypto` built without the `--with-openssl` option supports SM3/SM4 algorithms but not DES/3DES/CAST5. The detailed algorithm support is shown in the table below:

Encryption Algorithm	Without OpenSSL	With OpenSSL
MD5	Supported	Supported
SHA1	Supported	Supported
SHA224/256/384/512	Supported	Supported
Other Digest Algorithms	Not Supported	Supported
Blowfish	Supported	Supported
AES	Supported	Supported
DES/3DES/CAST5	Not Supported	Supported
Row Encryption	Supported	Supported
PGP Symmetric Encryption	Supported	Supported
PGP Public-Key Encryption	Supported	Supported
SM3	Supported	Not Supported
SM4	Supported	Not Supported

7.2.2 General hashing functions

`digest()`

```
digest(data text, type text) returns bytea
digest(data bytea, type text) returns bytea
```

This function computes and returns the binary hash value of `data`. `type` specifies the algorithm

to use. Standard algorithms include `md5`, `sha1`, `sha224`, `sha256`, `sha384`, and `sha512`. Additionally, any digest algorithm supported by OpenSSL will be available.

If you are using a non-OpenSSL version of `pgcrypto`, the `type` parameter will also support `sm3`.

To return the result as a hexadecimal string, use `encode()`. For example:

```
CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
    SELECT encode(digest($1, 'sha1'), 'hex')
$$ LANGUAGE SQL STRICT IMMUTABLE;
```

`hmac()`

```
hmac(data text, key text, type text) returns bytea
hmac(data bytea, key bytea, type text) returns bytea
```

This function computes the MAC value of `data` using the `key`. The `type` parameter uses the same standard algorithms as `digest()`.

Similar to `digest()`, this function allows hash value recalculation only if you know the key. This prevents someone from modifying the data and the hash value to match the key.

If the key is larger than the hash block size, it will be hashed first, and the resulting hash value will be used as the key.

7.2.3 Password hashing functions

The functions `crypt()` and `gen_salt()` are specifically designed for hashing passwords. `crypt()` performs the hashing, while `gen_salt()` prepares the algorithm parameters for it.

The algorithms in `crypt()` differ from typical MD5 or SHA1 hashing algorithms in the following ways:

- The algorithms in `crypt()` are intentionally slow. This is the only way to make brute-force password cracking difficult, given the small amount of data processed in batches.
- The algorithms in `crypt()` use a random value called "salt," so users with the same password will have different hashed passwords. This also provides additional defense against collision attacks.
- The algorithms in `crypt()` include the algorithm type in the result, allowing passwords hashed with different algorithms to coexist.

- Some algorithms in `crypt()` are adaptive. This means that as computers get faster, you can make the algorithm slower without introducing incompatibilities with existing passwords.

The table below lists the algorithms supported by the `crypt()` function:

Algorithm	Max password length	Adaptive	Salt bits	Output length	Description
bf	72	yes	128	60	Based on Blowfish, 2a variant
md5	Unlimited	no	48	34	MD5-based encryption
xdes	8	yes	24	20	Extended DES
des	8	no	12	13	Original UNIX encryption algorithm

crypt()

```
crypt(password text, salt text) returns text
```

This function computes the crypt(3)-style hash of the `password`. When storing a new password, you need to generate a new `salt` value using `gen_salt()`. To check a password, pass the stored hash as the `salt` and test if the result matches the stored value.

Example of setting a new password:

```
UPDATE ... SET pswhash = crypt('new password', gen_salt('md5'));
```

Example of authentication:

```
SELECT (pswhash = crypt('entered password', pswhash)) AS pswmatch FROM ... ;
```

This returns `true` if the entered password is correct.

gen_salt()

```
gen_salt(type text [, iter_count integer ]) returns text
```

This function generates a new random salt value for use with `crypt()`, specifying which hashing algorithm to use.

The `type` parameter determines the hashing algorithm. Supported types include `des`, `xdes`, `md5`, and `bf`.

For algorithms supporting an iteration count (e.g., `bf`), the `iter_count` parameter sets the number of iterations. A higher count increases the time required to hash and crack the password.

If omitted, a default value is used. Allowed values depend on the algorithm:

Algorithm	Default	Min	Max
xdes	725	1	16777215
bf	6	4	31

For `xdes`, the iteration count must be odd.

When choosing an iteration count, consider that the original DES crypt aimed for 4 hashes per second on contemporary hardware. Below 4 hashes per second may degrade usability, while above 100 hashes per second may be too fast.

The table below shows the relative slowness of different hashing algorithms, assuming an 8-character input with lowercase letters or a mix of uppercase, lowercase, and digits. The `crypt-bf` entries include the `iter_count` parameter after the slash.

Algorithm	Hashes/Secor	Lowercase letters `[a-z]`	Uppercase, lowercase, and digits `[A-Za-z0-9]`	Factor relative to MD5 hash speed
`crypt-bf/`	1792	4 years	3927 years	100k
`crypt-bf/`	3648	2 years	1929 years	50k
`crypt-bf/`	7168	1 year	982 years	25k
`crypt-bf/`	13504	188 years	521 years	12.5k
`crypt-md5`	171584	15 days	41 years	1k
`crypt-des`	23221568	157.5 minutes	108 days	7
`sha1`	37774272	90 minutes	68 days	4
`md5 (hash)`	150085504	22.5 minutes	17 days	1

Attention

- Tests were conducted on an Intel Mobile Core i3 processor.
- `crypt-des` and `crypt-md5` numbers are from John the Ripper v1.6.38 `-test` output.
- `md5` hash numbers are from mdcrack 1.2.
- `sha1` numbers are from lcrack-20031130-beta.
- `crypt-bf` numbers were measured using a simple program looping over 1000 8-character passwords.

Note: "Trying all combinations" is impractical. Password cracking typically uses dictionaries containing common words and specific character sets. Thus, passwords resembling words may be cracked faster than shown above, while non-word passwords might be uncrackable—or not.

7.2.4 PGP encryption functions

The PGP encryption functions implement the encryption part of the OpenPGP standard (RFC4880). These functions support both symmetric-key and public-key encryption.

Encrypted PGP messages consist of two parts or "packets":

- A packet containing the session key—encrypted using either a symmetric key or a public key.
- A packet containing the data encrypted with the session key.

When using symmetric-key (that is, password-based) encryption:

- The provided password is hashed using the `String2Key` (S2K) algorithm, which is similar to the `crypt()` algorithm (intentionally slow and salted) but generates a full-length binary key.
- If a separate session key is requested, a new random key is generated. Otherwise, the S2K key is used directly as the session key.
- If the S2K key is used directly, only the S2K settings are included in the session key packet. Otherwise, the session key is encrypted with the S2K key and placed in the session key packet.

When using public-key encryption:

- A new random session key is generated.
- The session key is encrypted using the public key and placed in the session key packet.

In either case, the data to be encrypted is processed as follows:

- Optional data manipulation: compression, conversion to UTF-8, and/or newline conversion.
- The data is prefixed with a block of random bytes, equivalent to using a random IV.
- An SHA1 hash is appended as a random prefix.
- All data encrypted with the session key is wrapped in a packet.

pgp_sym_encrypt()

```
pgp_sym_encrypt(data text, psw text [, options text ]) returns bytea
pgp_sym_encrypt_bytea(data bytea, psw text [, options text ]) returns bytea
```

Encrypts `data` using the symmetric PGP key `psw`. The `options` parameter can include settings as described below.

pgp_sym_decrypt()

```
pgp_sym_decrypt(msg bytea, psw text [, options text ]) returns text
pgp_sym_decrypt_bytea(msg bytea, psw text [, options text ]) returns bytea
```

Decrypts a PGP message encrypted with a symmetric key. To avoid outputting invalid character data, use `pgp_sym_decrypt_bytea` to decrypt raw text data.

pgp_pub_encrypt()

```
pgp_pub_encrypt(data text, key bytea [, options text ]) returns bytea
pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ]) returns bytea
```

Encrypts `data` using the public PGP key `key`. Providing a private key to this function will result in an error.

pgp_pub_decrypt()

```
pgp_pub_decrypt(msg bytea, key bytea [, psw text [, options text ]]) returns text
pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text [, options text ]]) returns
bytea
```

Decrypts a message encrypted with a public key. The `key` must be the private key corresponding to the public key used for encryption. If the key is password-protected, the password must be provided via `psw`. If there is no password, provide an empty password as a placeholder. To avoid outputting invalid character data, use `pgp_pub_decrypt_bytea` to decrypt raw text data.

pgp_key_id()

```
pgp_key_id(bytea) returns text
```

The `pgp_key_id` function extracts the key ID from a PGP public or private key. If given an encrypted message, it returns the key ID used for encryption.

This function can return two special key IDs:

- `SYMKEY`: The message is encrypted with a symmetric key.
- `ANYKEY`: The message is encrypted with a public key, but the key ID has been removed. To decrypt it, you need to try all available keys. Note that `pgcrypto` does not generate such messages.

Different keys may share the same ID, though this is rare. In such cases, client applications should attempt decryption with each key to determine which one works, similar to handling `ANYKEY`.

armor(), dearmor()

```
armor(data bytea [ , keys text[], values text[] ]) returns text
dearmor(data text) returns bytea
```

These functions convert binary data to/from the PGP ASCII-armor format, which is essentially Base64 with CRC and additional formatting.

If the `keys` and `values` arrays are specified, an armor header is added for each key/value pair. Both arrays must be one-dimensional and of the same length, and must not contain non-ASCII characters.

pgp_armor_headers

```
pgp_armor_headers(data text, key out text, value out text) returns setof record
```

The `pgp_armor_headers` function extracts armor headers from the given data. The return value is a set of rows with two columns: `key` and `value`. Non-ASCII characters in keys or values are treated as UTF-8.

Options for PGP functions

PGP function options are similar to those in GnuPG. Values are specified after an equals sign, with multiple options separated by commas. For example:

```
pgp_sym_encrypt(data, psw, 'compress-algo=1, cipher-algo=aes256')
```

Most options apply only to encryption functions, as decryption functions obtain parameters directly from the PGP data. Key options include `compress-algo` and `unicode-mode`, while others have sensible defaults.

Here are the available options:

cipher-algo

- Specifies the cipher algorithm to use.
- Possible values: bf, aes128, aes192, aes256, 3des, cast5
- Default value: aes128
- Applies to: `pgp_sym_encrypt`, `pgp_pub_encrypt`

compress-algo

- Specifies the compression algorithm. Requires SynxDB Elastic to be built with zlib.
- Possible values:
  - `0`: No compression
  - `1`: ZIP compression
  - `2`: ZLIB compression (ZIP with metadata and block CRC)
- Default value: 0
- Applies to: `pgp_sym_encrypt`, `pgp_pub_encrypt`

compress-level

- Specifies the compression level. Higher levels improve compression but slow down processing. `0` disables compression.
- Possible values: `0`, `1-9`
- Default value: 6
- Applies to: `pgp_sym_encrypt`, `pgp_pub_encrypt`

convert-crlf

- Converts `\n` to `\r\n` during encryption and `\r\n` to `\n` during decryption. Use this option for full RFC4880 compliance, as text data should use `\r\n` line endings.
- Possible values: `0`, `1`
- Default value: `0`
- Applies to: `pgp_sym_encrypt`, `pgp_pub_encrypt`, `pgp_sym_decrypt`, `pgp_pub_decrypt`

disable-mdc

- Disables SHA-1 protection for data. Use this option only for compatibility with older PGP products that do not support SHA-1 protection packets (added in RFC 4880). Modern software like gnupg.org and pgp.com supports this feature.
- Possible values: `0`, `1`
- Default value: 0
- Applies to: `pgp_sym_encrypt`, `pgp_pub_encrypt`

sess-key

- Use a separate session key for symmetric-key encryption. Public-key encryption always uses a separate session key. This option defaults to directly using the S2K key.
- Values: 0, 1
- Default: 0

- Applies to: pgp_sym_encrypt

s2k-mode

- Specifies which S2K algorithm to use.
- Optional values:
  - 0: No salt (dangerous, use with caution).
  - 1: Salt with a fixed iteration count.
  - 3: Salt with a variable iteration count.
- Default: 3
- Applies to: pgp_sym_encrypt

s2k-count

- The iteration count for the S2K algorithm, which must be between `1024` and `65011712`.
- Default: A random value between `65536` and `253952`.
- Applies to: pgp_sym_encrypt (only if s2k-mode=3)

s2k-digest-algo

- Specifies which digest algorithm to use in S2K calculations.
- Optional values: md5, sha1
- Default: sha1
- Applies to: pgp_sym_encrypt

s2k-cipher-algo

- Specifies which cipher to use for encrypting the separate session key.
- Optional values: bf, aes, aes128, aes192, aes256
- Default: The cipher algorithm used
- Applies to: pgp_sym_encrypt

unicode-mode

- Whether to convert text data from the internal database encoding to UTF-8 and back. If your database is already in UTF-8, no conversion will occur, but the message will be marked as UTF-8. Without this option, no conversion is performed.
- Optional values: 0, 1
- Default: 0
- Applies to: pgp_sym_encrypt, pgp_pub_encrypt

Generate PGP keys with GnuPG

- Generate a new key:

```
gpg --gen-key
```

  The preferred key type is "DSA" and "Elgamal". For RSA encryption, you must create a DSA or RSA signing key as the master key, then add an RSA encryption subkey using `gpg --edit-key`.

- List keys:

```
gpg --list-secret-keys
```

- Export the public key in ASCII armor format:

```
gpg -a --export KEYID > public.key
```

- Export the key in ASCII armor format:

```
gpg -a --export-secret-keys KEYID > secret.key
```

Before passing these keys to PGP functions, you need to use `dearmor()` on them. Alternatively, if you can handle binary data, you can remove the `-a` from the command.

For more details, refer to `man gpg`, the GNU Privacy Handbook, and other GnuPG documentation.

Limitations of PGP code

- Multiple subkeys are not supported. This may seem like an issue since subkeys are a common practice. On the other hand, you should not use regular GPG or PGP keys with `pgcrypto`. Instead, create new keys, as the use cases are quite different.

7.2.5 Raw encryption functions

These functions apply the cipher directly to the data and do not include any of the advanced features of PGP encryption. Therefore, the main issues are:

- The user's key is used directly as the cipher key.
- No integrity checks are provided; you cannot tell if the encrypted data has been tampered with.
- The user must manage all encryption parameters, including the IV, themselves.
- Text handling is not supported.

Given the introduction of PGP encryption, the use of raw encryption functions is discouraged.

```
encrypt(data bytea, key bytea, type text) returns bytea
decrypt(data bytea, key bytea, type text) returns bytea

encrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
decrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
```

Encrypt or decrypt data using the cipher method specified by `type`. The syntax for the `type` string is:

```
**algorithm** [ - **mode** ] [ /pad: **padding** ]
```

Where **algorithm** is one of the following:

- `BF` - Blowfish
- `AES` - AES (Rijndael-128, -192, or -256)
    - If you are using a non-OpenSSL version of `pgcrypto`, `sm4` will be an optional value.

mode is one of the following:

- `CBC` - Each block depends on the previous one (default).
- `ECB` - Each block is encrypted separately (not secure and not recommended).

`padding` is one of the following:

- `pkcs` - Data can be of any length (default).
- `none` - Data must be a multiple of the cipher block size.

For example, these are equivalent:

```
encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')
```

In `encrypt_iv` and `decrypt_iv`, the `iv` parameter is the initial value for CBC mode; it is ignored for ECB. If not a full block size, it will be clipped or padded with zeros. In functions without this parameter, it defaults to all zeros.

7.2.6 Random data functions

```
gen_random_bytes(count integer) returns bytea
```

Returns `count` cryptographically strong random bytes. Up to 1024 bytes can be extracted at once to avoid depleting the random generator pool.

```
gen_random_uuid() returns uuid
```

Returns a version 4 (random) UUID. (Deprecated; this function internally calls the core function of the same name.)

7.2.7 Notes

Configuration

`pgcrypto` configures itself based on the discoveries of the main SynxDB Elastic `configure` script. The options that affect it are `--with-zlib` and `--with-openssl`.

When compiled with zlib, the PGP encryption functions can compress data before encryption.

When compiled against OpenSSL 3.0.0 or higher, the legacy provider must be activated in the `openssl.cnf` configuration file to use older encryption methods like DES or Blowfish.

NULL handling

In accordance with the SQL standard, all functions return NULL if any argument is NULL. This can pose a security risk if not used carefully.

Security limitations

All `pgcrypto` functions run inside the database server. This means that all data and passwords are transmitted in plain text between `pgcrypto` and the client application. Therefore, you should:

1. Use local connections or SSL connections.
2. Trust the system and database administrators.

If this is not possible, it is better to perform encryption in the client application.

`pgcrypto` is not resistant to side-channel attacks. For example, the time taken by the `pgcrypto` decryption function varies depending on the size of the ciphertext.

7.3 Log Auditing pgAudit

The pgAudit extension provides detailed session or object audit logging through PostgreSQL's standard logging functionality.

pgAudit enables users to generate audit logs, which are often required for government, financial, or ISO certifications. Auditing is an official inspection of an individual's or organization's accounts, typically conducted by an independent body.

pgAudit is pre-installed in SynxDB Elastic. When using pgAudit, ORCA must be disabled.

```
SET optimizer = off;
```

7.3.1 Configure pgAudit

Only superusers can modify pgAudit settings. Allowing ordinary users to change these settings would compromise the integrity of the audit logs.

Settings can be configured at the global level (in `postgresql.conf` or using `ALTER SYSTEM ... SET`), database level (using `ALTER DATABASE ... SET`), or role level (using `ALTER ROLE ... SET`). Note that these settings cannot be inherited through ordinary roles, and `SET ROLE` will not change a user's pgAudit settings. This is a limitation of the role system, not pgAudit itself.

The pgAudit extension must be loaded in shared_preload_libraries. Otherwise, an error will occur at load time, and audit logging will not function.

Before setting `pgaudit.log_class`, you must first call `CREATE EXTENSION pgaudit`. This extension installs event triggers that provide additional audit functionality for DDL operations. pgAudit will still work without the extension, but DDL statements will lack information about object types and names.

If you need to drop and recreate the `pgaudit` extension, you must first remove the `pgaudit.log_class` setting; otherwise, an error will occur.

pgaudit.log_class

Specifies the categories of statements to be logged for session audits. Possible values include:

- READ: `SELECT` and `COPY` when the source is a relation or query.
- WRITE: `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, and `COPY` when the target is a relation.

- FUNCTION: Function calls and `DO` blocks.
- ROLE: Role and permission-related statements, such as `GRANT`, `REVOKE`, and `CREATE/ALTER/DROP ROLE`.
- DDL: All `DDL` statements not included in the `ROLE` category.
- MISC: Other commands, such as `DISCARD`, `FETCH`, `CHECKPOINT`, `VACUUM`, and `SET`.
- MISC_SET: Other `SET` commands, such as `SET ROLE`.
- ALL: All of the above.

Multiple categories can be specified in a comma-separated list, and categories can be excluded by prefixing them with a `-` symbol. The default value is `none`.

pgaudit.log_catalog

- Enables session logging when all relations in a statement are within `pg_catalog`. Disabling this reduces log noise from tools like psql and pgAdmin that perform many catalog queries.
- Default setting is `on`.

pgaudit.log_client

- Determines whether log messages are visible to client processes (e.g., psql). This should generally remain disabled but may be useful for debugging or other purposes.
- Note that the `pgaudit.log` level is only enabled when `pgaudit.log_client` is turned on; otherwise, the default value is used.
- Default setting is `off`.

pgaudit.log_level

- Specifies the log level for audit log entries. Note that `ERROR`, `FATAL`, and `PANIC` are not allowed. This setting is primarily used for regression testing and may also be useful for end-user testing or other purposes.
- Note that the `pgaudit.log` level is only enabled when `pgaudit.log_client` is turned on; otherwise, the default value is used.
- Default is `log`.

pgaudit.log_parameter

- Determines whether audit logs should include parameters passed with statements. If enabled, parameters will be included in CSV format after the statement text.
- Default setting is `off`.

pgaudit.log_relation

- Determines whether session audit logs should create separate entries for each relation (e.g., `TABLE`, `VIEW`) referenced in `SELECT` or DML statements. This provides a useful shortcut for detailed logging without using object-level auditing.
- Default setting is `off`.

pgaudit.log_rows

- Determines whether audit logs should include the number of rows retrieved or affected by a statement. If enabled, the row count will be included after the parameter field.
- Default setting is `off`.

pgaudit.log_statement

- Indicates whether the statement text and parameters (if enabled) are included in the log. Depending on the requirements, this level of detail may not be necessary for audit logs.
- Default setting is `on`.

pgaudit.log_statement_once

- Determines whether the statement text and parameters are logged only with the first entry for a statement/sub-statement combination or with each entry. Enabling this reduces log verbosity but may complicate identifying the statement that generated the log entry. However, the statement/sub-statement and process ID should still allow identification of the statement text from previous entries.
- Default setting is `off`.

pgaudit.role

- Specifies the primary role used for object audit logging. Multiple audit roles can be defined by granting them to this primary role, allowing different groups to manage various aspects of audit logging.
- There is no default value.

7.3.2 Session Audit Logging

Session audit logging provides a detailed record of all statements executed by the backend on behalf of the user.

Configuration

Session logging is enabled via the `pgaudit.log_class` setting.

Enable session logging for all DML and DDL, and log all relations in DML statements:

```
SET pgaudit.log_class = 'write, ddl';
SET pgaudit.log_relation = on;
```

Enable session logging for all commands except `MISC`, and set audit log messages to `NOTICE`:

```
SET pgaudit.log_class = 'all, -misc';
SET pgaudit.log_level = notice;
```

Examples

In this example, session audit logging is used to log DDL and `SELECT` statements. Note that `INSERT` statements are not logged because the `WRITE` category is not enabled.

```
SET pgaudit.log_class = 'read, ddl';

CREATE TABLE account
(
    id INT,
    name TEXT,
    password TEXT,
    description TEXT
);
```

(continues on next page)

(continued from previous page)

```
INSERT INTO account (id, name, password, description)
             VALUES (1, 'user1', 'HASH1', 'blah, blah');

SELECT *
    FROM account;
```

Log output:

```
AUDIT: SESSION,1,1,DDL,CREATE TABLE,TABLE,public.account,CREATE TABLE account
(
    id INT,
    name TEXT,
    password TEXT,
    description TEXT
);,<not logged>
AUDIT: SESSION,2,1,READ,SELECT,,,SELECT *
    FROM account,,<not logged>
```

7.3.3 Object audit logging

Object audit logging captures statements affecting specific relations, supporting only `SELECT`, `INSERT`, `UPDATE`, and `DELETE` commands. `TRUNCATE` is excluded from object audit logging.

Object audit logging offers a more granular alternative to setting `pgaudit.log_class = 'read, write'`. Combining them is generally not recommended, but one use case could be to first capture all statements via session logging and then enhance the logs with object-level details for specific relations.

Configuration

Object-level audit logging is managed through the role system. The `pgaudit.role` parameter specifies the role used for audit logging. When this role has permissions to execute a command or inherits permissions from another role, audit logs for the affected relation (such as `TABLE`, `VIEW`, etc.) will be recorded. This approach allows multiple audit roles to be defined, even though only one primary role is used in any context.

To enable object audit logging, set `pgaudit.role` to `auditor` and grant `SELECT` and `DELETE` permissions on the `account` table. Any `SELECT` or `DELETE` statements on this table will be

logged:

```
SET pgaudit.role = 'auditor';

GRANT SELECT, DELETE
   ON public.account
   TO auditor;
```

Example

This example demonstrates how object audit logging can be used for fine-grained logging of `SELECT` and DML statements. Note that logging for the `account` table is controlled by column-level permissions, while logging for the `account_role_map` table is controlled by table-level permissions.

```
set pgaudit.role = 'auditor';

create table account
(
    id int,
    name text,
    password text,
    description text
);

grant select (password)
   on public.account
   to auditor;

select id, name
  from account;

select password
  from account;

grant update (name, password)
   on public.account
   to auditor;

update account
   set description = 'yada, yada';
```

(continues on next page)

(continued from previous page)

```
update account
   set password = 'HASH2';

create table account_role_map
(
    account_id int,
    role_id int
);

grant select
   on public.account_role_map
   to auditor;

select account.password,
       account_role_map.role_id
  from account
       inner join account_role_map
            on account.id = account_role_map.account_id
```

Log output:

```
AUDIT: OBJECT,1,1,READ,SELECT,TABLE,public.account,select password
  from account,<not logged>
AUDIT: OBJECT,2,1,WRITE,UPDATE,TABLE,public.account,update account
   set password = 'HASH2',<not logged>
AUDIT: OBJECT,3,1,READ,SELECT,TABLE,public.account,select account.password,
       account_role_map.role_id
  from account
       inner join account_role_map
            on account.id = account_role_map.account_id,<not logged>
AUDIT: OBJECT,3,1,READ,SELECT,TABLE,public.account_role_map,select account.password,
       account_role_map.role_id
  from account
       inner join account_role_map
            on account.id = account_role_map.account_id,<not logged>
```

7.3.4 Format

Audit entries are written to standard logging tools in a comma-separated format. The output conforms to CSV format only if the log line prefix is removed from each log item.

- **AUDIT_TYPE**: `SESSION` or `OBJECT`.
- **STATEMENT_ID**: A unique identifier for each statement in the session. Statement IDs are consecutive, even if some statements are not logged. A single statement ID may have multiple entries when logging multiple relations.
- **SUBSTATEMENT_ID**: A sequential identifier for each sub-statement within the main statement (e.g., functions called from a query). Sub-statement IDs are consecutive, even if some sub-statements are not logged. A single sub-statement ID may have multiple entries when logging multiple relations.
- **CLASS**: For example, `READ`, `ROLE`.
- **COMMAND**: For example, `ALTER TABLE`, `SELECT`.
- **OBJECT_TYPE**: For example, `TABLE`, `VIEW`. Applies to `SELECT`, DML, and most DDL statements.
- **OBJECT_NAME**: The fully qualified object name (e.g., `public.account`). Applies to `SELECT`, DML, and most DDL statements.
- **STATEMENT**: The statement executed by the backend.
- **PARAMETER**: If `pgaudit.log_parameter` is enabled, this field contains the statement parameters in CSV format, or `<none>` if there are no parameters. Otherwise, the field is `<not logged>`.

Use log_line_prefix to add additional fields required for audit log compliance. A typical log line prefix might be `'%m %u %d [%p]: '`, which includes the date/time, username, database name, and process ID for each audit log entry.

7.4 Manage Roles and Privileges

The SynxDB Elastic authorization mechanism stores roles and privileges to access database objects in the database and is administered using SQL statements or command-line utilities.

SynxDB Elastic manages database access privileges using *roles*. The concept of roles subsumes the concepts of *users* and *groups*. A role can be a database user, a group, or both. Roles can own database objects (for example, tables) and can assign privileges on those objects to other roles to control access to the objects. Roles can be members of other roles, thus a member role can inherit the object privileges of its parent role.

Every SynxDB Elastic system contains a set of database roles (users and groups). Those roles are separate from the users and groups managed by the operating system on which the server runs. However, for convenience you might want to maintain a relationship between operating system user names and SynxDB Elastic role names, since many of the client applications use the current operating system user name as the default.

In SynxDB Elastic, users log in and connect through the coordinator instance, which then verifies their role and access privileges. The coordinator then issues commands to the segment instances behind the scenes as the currently logged-in role.

Roles are defined at the system level, meaning they are valid for all databases in the system.

To bootstrap the SynxDB Elastic system, a freshly initialized system always contains one predefined *superuser* role (also referred to as the system user). This role will have the same name as the operating system user that initialized the SynxDB Elastic system. Customarily, this role is named `gpadmin`. In order to create more roles you first have to connect as this initial role.

7.4.1 Create new roles (users)

A user-level role is considered to be a database role that can log in to the database and initiate a database session. Therefore, when you create a new user-level role using the `CREATE ROLE` command, you must specify the `LOGIN` privilege. For example:

```
CREATE ROLE jsmith WITH LOGIN;
```

A database role might have a number of attributes that define what sort of tasks that role can perform in the database. You can set these attributes when you create the role, or later using the `ALTER ROLE` command.

Alter role attributes

A database role might have a number of attributes that define what sort of tasks that role can perform in the database.

Attributes	Description
`SUPERUSER` or `NOSUPERUSER`	Determines if the role is a superuser. You must yourself be a superuser to create a new superuser. `NOSUPERUSER` is the default.
`CREATEDB` or `NOCREATEDB`	Determines if the role is allowed to create databases. `NOCREATEDB` is the default.
`CREATEROLE` or `NOCREATEROLE`	Determines if the role is allowed to create and manage other roles. `NOCREATEROLE` is the default.
`INHERIT` or `NOINHERIT`	Determines whether a role inherits the privileges of roles it is a member. A role with the `INHERIT` attribute can automatically use whatever database privileges have been granted to all roles it is directly or indirectly a member of. `INHERIT` is the default.
`LOGIN` or `NOLOGIN`	Determines whether a role is allowed to log in. A role having the `LOGIN` attribute can be thought of as a user. Roles without this attribute are useful for managing database privileges (groups). `NOLOGIN` is the default.
`CONNECTION LIMIT *connlimit*`	If role can log in, this specifies how many concurrent connections the role can make. -1 (the default) means no limit.
`CREATEEXTTABLE` or `NOCREATEEXTTABLE`	Determines whether a role is allowed to create external tables. `NOCREATEEXTTABLE` is the default. For a role with the `CREATEEXTTABLE` attribute, the default external table `type` is `readable` and the default `protocol` is `gpfdist`. Note that external tables that use the `file` protocol can only be created by superusers.
`PASSWORD '*password*'`	Sets the role's password. If you do not plan to use password authentication you can omit this option. If no password is specified, the password will be set to null and password authentication will always fail for that user. A null password can optionally be written explicitly as `PASSWORD NULL`.
`DENY deny_interval` or `DENY deny_point`	Restricts access during an interval, specified by day or day and time.

You can set these attributes when you create the role, or later using the `ALTER ROLE` command. For example:

```
ALTER ROLE jsmith WITH PASSWORD 'passwd123';
ALTER ROLE jsmith LOGIN;
ALTER ROLE jsmith DENY DAY 'Sunday';
```

A role can also have role-specific defaults for many of the server configuration settings. For example, to set the default schema search path for a role:

```
ALTER ROLE admin SET search_path TO myschema, public;
```

7.4.2 Role membership

It is frequently convenient to group users together to ease management of object privileges: that way, privileges can be granted to, or revoked from, a group as a whole. In SynxDB Elastic, this is done by creating a role that represents the group, and then granting membership in the group role to individual user roles.

Use the `CREATE ROLE` SQL command to create a new group role. For example:

```
CREATE ROLE admin CREATEROLE CREATEDB;
```

Once the group role exists, you can add and remove members (user roles) using the `GRANT` and `REVOKE` commands. For example:

```
GRANT admin TO john, sally;
REVOKE admin FROM bob;
```

For managing object privileges, you would then grant the appropriate privileges to the group-level role only (see *Manage object privileges*). The member user roles then inherit the object privileges of the group role. For example:

```
GRANT ALL ON TABLE mytable TO admin;
GRANT ALL ON SCHEMA myschema TO admin;
GRANT ALL ON DATABASE mydb TO admin;
```

The role attributes `LOGIN`, `SUPERUSER`, `CREATEDB`, `CREATEROLE`, `CREATEEXTTABLE`, and `RESOURCE QUEUE` are never inherited as ordinary privileges on database objects are. User members must actually `SET ROLE` to a specific role having one of these attributes in order to make use of the attribute. In the above example, we gave `CREATEDB` and `CREATEROLE` to the `admin` role. If `sally` is a member of `admin`, then `sally` could issue the following command to assume the role attributes of the parent role:

```
SET ROLE admin;
```

7.4.3 Manage object privileges

When an object (table, view, sequence, database, function, language, schema, or tablespace) is created, it is assigned an owner. The owner is normally the role that ran the creation statement. For most kinds of objects, the initial state is that only the owner (or a superuser) can do anything with the object. To allow other roles to use it, privileges must be granted. SynxDB Elastic supports the following privileges for each object type:

Object Type	Privileges
Tables, External Tables, Views	`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `REFERENCES`, `TRIGGER`, `TRUNCATE`, `ALL`
Columns	`SELECT`, `INSERT`, `UPDATE`, `REFERENCES`, `ALL`
Sequences	`USAGE`, `SELECT`, `UPDATE`, `ALL`
Databases	`CREATE`, `CONNECT`, `TEMPORARY`, `TEMP`, `ALL`
Domains	`USAGE`, `ALL`
Foreign Data Wrappers	`USAGE`, `ALL`
Foreign Servers	`USAGE`, `ALL`
Functions	`EXECUTE`, `ALL`
Procedural Languages	`USAGE`, `ALL`
Schemas	`CREATE`, `USAGE`, `ALL`
Tablespaces	`CREATE`, `ALL`
Types	`USAGE`, `ALL`
Protocols	`SELECT`, `INSERT`, `ALL`

Note

You must grant privileges for each object individually. For example, granting `ALL` on a database does not grant full access to the objects within that database. It only grants all of the database-level privileges (`CONNECT`, `CREATE`, `TEMPORARY`) to the database itself.

Use the `GRANT` SQL command to give a specified role privileges on an object. For example, to grant the role named `jsmith` insert privileges on the table named `mytable`:

```
GRANT INSERT ON mytable TO jsmith;
```

Similarly, to grant `jsmith` select privileges only to the column named `col1` in the table named `table2`:

```
GRANT SELECT (col1) on TABLE table2 TO jsmith;
```

To revoke privileges, use the `REVOKE` command. For example:

```
REVOKE ALL PRIVILEGES ON mytable FROM jsmith;
```

You can also use the `DROP OWNED` and `REASSIGN OWNED` commands for managing objects owned by deprecated roles (Note: only an object's owner or a superuser can drop an object or reassign ownership). For example:

```
REASSIGN OWNED BY sally TO bob;
DROP OWNED BY visitor;
```

7.4.4 Security best practices for roles and privileges

- **Secure the gpadmin system user.**

  SynxDB Elastic requires a UNIX user ID to install and initialize the SynxDB Elastic system. This system user is referred to as `gpadmin` in the SynxDB Elastic documentation. This `gpadmin` user is the default database superuser in SynxDB Elastic, as well as the file system owner of the SynxDB Elastic installation and its underlying data files. This default administrator account is fundamental to the design of SynxDB Elastic. The system cannot run without it, and there is no way to limit the access of this gpadmin user ID.

  Use roles to manage who has access to the database for specific purposes. You should only use the `gpadmin` account for system maintenance tasks such as expansion and upgrade. Anyone who logs on to a SynxDB Elastic host as this user ID can read, alter or delete any data, including system catalog data and database access rights. Therefore, it is very important to secure the gpadmin user ID and only provide access to essential system administrators. Administrators should only log in to SynxDB Elastic as `gpadmin` when performing certain system maintenance tasks (such as upgrade or expansion). Database users should never log on as `gpadmin`, and ETL or production workloads should never run as `gpadmin`.

- **Assign a distinct role to each user that logs in.**

  For logging and auditing purposes, each user that is allowed to log in to SynxDB Elastic should be given their own database role. For applications or web services, consider creating a distinct role for each application or service. See *Create New Roles (Users)*.

- **Use groups to manage access privileges.**

  See *Role membership*.

- **Limit users who have the SUPERUSER role attribute.**

  Roles that are superusers bypass all access privilege checks in SynxDB Elastic, as well as resource queuing. Only system administrators should be given superuser rights. See *Alter Role Attributes*.

7.4.5 Encrypt data

SynxDB Elastic is installed with an optional module of encryption/decryption functions called `pgcrypto`. The `pgcrypto` functions allow database administrators to store certain columns of data in encrypted form. This adds an extra layer of protection for sensitive data, as data stored in SynxDB Elastic in encrypted form cannot be read by anyone who does not have the encryption key, nor can it be read directly from the disks.

> **Note**
>
> The `pgcrypto` functions run inside the database server, which means that all the data and passwords move between `pgcrypto` and the client application in clear-text.

To use `pgcrypto` functions, register the `pgcrypto` extension in each database in which you want to use the functions. For example:

```
CREATE EXTENSION pgcrypto;
```

7.5 Transparent Data Encryption

To meet the need for user data security, SynxDB Elastic supports Transparent Data Encryption (TDE).

Transparent Data Encryption (TDE) is a technology for encrypting database data files:

- **Data**: Refers to the data within the database.
- **Encryption at rest**: Data files are stored in an encrypted format on disk and are decrypted when read into memory. TDE protects data at rest.
- **Transparency**: The encryption and decryption processes are managed automatically by TDE, so users and applications can use it without changing their operational habits or code.

7.5.1 Introduction to encryption algorithms

Basic concepts

- **DEK (Data Encryption Key)**: Used to directly encrypt data. It is generated by the database and stored in memory.
- **DEK plaintext**: Synonymous with DEK, it can only be stored in memory.
- **Master Key**: The primary key used to encrypt the DEK, ensuring its security.
- **DEK ciphertext**: The DEK encrypted by the Master Key, which is persisted as ciphertext.

Key management module

The key management module is the core component of TDE. It uses a two-tier key structure with a Master Key and a DEK. In the cloud-native architecture of SynxDB Elastic, the Master Key is managed by an independent and secure `key-service` component deployed in a Kubernetes cluster. The DEK is used to encrypt database data, and its ciphertext is stored within the database.

Algorithm categories

Encryption algorithms are divided into symmetric and asymmetric encryption. Due to performance advantages, TDE primarily uses block cipher algorithms from symmetric encryption. SynxDB Elastic supports two industry-standard block cipher algorithms: AES and SM4.

AES encryption algorithm

AES is an international standard block cipher algorithm that supports 128, 192, and 256-bit keys. Common encryption modes include:

- ECB: Electronic Codebook mode
- CBC: Cipher Block Chaining mode
- CFB: Cipher Feedback mode
- OFB: Output Feedback mode
- CTR: Counter mode

More ISO/IEC encryption algorithms

Other ISO/IEC algorithms include:

- ISO/IEC 14888-3/AMD1 (that is, SM2): An asymmetric encryption algorithm based on ECC, with better performance than RSA.
- ISO/IEC 10118-3:2018 (that is, SM3): A message digest algorithm similar to MD5, with a 256-bit output.
- ISO/IEC 18033-3:2010/AMD1:2021 (that is, SM4): A symmetric encryption algorithm for wireless LAN standards, supporting 128-bit keys and block lengths.

7.5.2 How to use TDE

In SynxDB Elastic, TDE is specified when creating a database, enabling database-level encryption.

Before using TDE, you need to:

- Have permission to execute `CREATE DATABASE`.
- Ensure your version of SynxDB Elastic supports TDE.
- Have access to the `kubectl` command-line tool to manage the `key-service` during verification.

When creating a database, use the `WITH ENCRYPTION_ENABLE` clause to enable TDE and specify the tablespace and encryption algorithm (for example, `aes` or `sm4`).

- Create an encrypted database using the AES algorithm:

```
CREATE DATABASE encryptdb_aes WITH ENCRYPTION_ENABLE 'aes' tablespace default_
global_synxdb_tablespace;
```

- Create an encrypted database using the SM4 algorithm:

```
CREATE DATABASE encryptdb_sm4 WITH ENCRYPTION_ENABLE 'sm4' tablespace default_
global_synxdb_tablespace;
```

Once created, all data stored in this database (including future tables and data) will be automatically encrypted.

7.5.3 How to verify

TDE is transparent to upper-layer applications. To verify that TDE can protect data security if the master key is lost, you can simulate a scenario where the `key-service` is unavailable.

1. **Create an encrypted database and write data to the database**.

   First, create a database with AES encryption and switch to the database.

```
-- Create an encrypted database named encryptdb1
CREATE DATABASE encryptdb1 WITH ENCRYPTION_ENABLE 'aes' tablespace default_global_
synxdb_tablespace;

-- Switch to the newly created database
\c encryptdb1
```

   To see detailed encryption and decryption logs in the following steps, you can enable the `debug_tde_print_encrypt_data` debug parameter. This is an optional step, mainly for demonstration and debugging purposes.

```
-- Sets the warehouse and enable the debug parameter.
SET warehouse TO wl;
SET debug_tde_print_encrypt_data = true;
```

   Next, create a table and insert data. At this point, you should see a notification in the terminal indicating the encryption operation.

```
-- Creates a test table.
encryptdb1=# create table t1(id int);
CREATE TABLE

-- Inserts data to trigger encryption.
encryptdb1=# INSERT INTO t1 SELECT generate_series(1,5);
NOTICE:  Encrypt data BLock, use encrypt algorithm: AES_256 (seg0 127.0.0.1:5433
pid=23938)
INSERT 0 5
```

2. **Simulate the loss of the Master Key service**.

   The Master Key is managed by the key-service component. You can use the `kubectl` command to scale the number of replicas for this service down to 0, simulating service unavailability.

```
# Scales the number of replicas for the key-service deployment to 0.
kubectl scale --replicas=0 deployment/key-service -n <your-namespace>
```

   Replace `<your-namespace>` with the namespace where your SynxDB Elastic system is running, for example, `synxdb-system-4x`.

3. **Verify that data is inaccessible when the key is lost**.

   Because key information is cached in the database, you need to restart the database for the `key-service` shutdown to take effect. After the database restarts, try to query the data in the encrypted table.

```
encryptdb1=# select * from t1;
ERROR:  get master key fail, dboid: 16397, key_path:/16397/1  (hd_keys_manager.
c:259)
```

   The query will fail and return a "get master key fail" error. This clearly indicates that the database cannot decrypt the data without access to the master key from the `key-service`, thus verifying the security of TDE.

4. **Restore the Master Key service and verify the data**.

```
# Restores the number of replicas for the key-service deployment to 1.
kubectl scale --replicas=1 deployment/key-service -n <your-namespace>
```

Restart the database again to clear the error state and reload the key. After the database restarts, data queries should return to normal.

```
-- Successfully query data, triggering decryption.
encryptdb1=# SELECT * FROM t1; [cite: 70]
NOTICE:  Decrypt data BLock, use encrypt algorithm: AES_256 (seg0 slice1 127.0.0.
1:5433 pid=23938)
id
----
(5 rows)
```

By following these steps, you can fully verify the TDE feature in SynxDB Elastic: it is transparent to users under normal conditions and effectively prevents data access when the key service is abnormal, ensuring the security of data at rest.

7.6 Data Anonymization Using Anon

Anon is an extension for SynxDB Elastic, adapted from PostgreSQL Anonymizer. It provides data anonymization features for SynxDB Elastic to prevent sensitive data leakage.

Data anonymization, also known as data masking, is the process of removing sensitive information from data. This includes personal names, phone numbers, addresses, and ID numbers. Anon uses the `SECURITY LABEL` feature from PostgreSQL to define masking rules and applies them to specific database objects. For example, an original phone number `0609110911` becomes `06******11` after anonymization.

7.6.1 How it works

Anon primarily uses PostgreSQL's `SECURITY LABEL` feature to store masking rules. For dynamic masking, when a role marked as a masked user accesses data, Anon rewrites the SQL query at the optimizer level, replacing sensitive data with masked data. This way, the database can return different data for different users, achieving data anonymization without creating extra views or modifying the original data.

7.6.2 Usage guide

Anon provides two data anonymization methods: static and dynamic.

Declare masking rules

The core concept of Anon is to provide "anonymization by design".

Masking rules should be written by application developers who best understand the data model. Therefore, the rules must be implemented directly within the database schema. This allows data to be anonymized directly inside the PostgreSQL instance without external tools, thus limiting the risk of data exposure and leakage.

Declare masking rules using `SECURITY LABEL`:

```
CREATE EXTENSION IF NOT EXISTS anon CASCADE;
SELECT anon.init();

CREATE TABLE player( id SERIAL, name TEXT, points INT);

SECURITY LABEL FOR anon ON COLUMN player.name
```

(continues on next page)

(continued from previous page)

```
IS 'MASKED WITH FUNCTION anon.fake_last_name()';

SECURITY LABEL FOR anon ON COLUMN player.id
IS 'MASKED WITH VALUE NULL';
```

Static masking

> **Warning**
>
> Anon also provides a static masking feature, but it is "dangerous" because it directly updates the database with masked data, causing the loss of original data. Therefore, it is not recommended to use this feature in a production environment.
>
> However, this feature is useful in testing scenarios, and many existing tests use static masking. The relevant functions are `anon.anonymize_database()`, `anon.anonymize_table()`, and `anon.anonymize_column()`.

You can use the `anon.anonymize_database()` function to permanently remove Personally Identifiable Information (PII) from the database. This will destroy the original data, so use it with caution.

For example:

```
-- Original data
SELECT * FROM customer;
 id  |   full_name      |   birth    |    employer    | zipcode | fk_shop
-----+------------------+------------+----------------+---------+---------
 911 | Chuck Norris     | 1940-03-10 | Texas Rangers | 75001   | 12
 112 | David Hasselhoff | 1952-07-17 | Baywatch      | 90001   | 423

-- Define masking rules
SECURITY LABEL FOR anon ON COLUMN customer.full_name
IS 'MASKED WITH FUNCTION anon.fake_first_name() || '' '' || anon.fake_last_name()';

SECURITY LABEL FOR anon ON COLUMN customer.birth
IS 'MASKED WITH FUNCTION anon.random_date_between(''1920-01-01''::DATE,now())';

SECURITY LABEL FOR anon ON COLUMN customer.employer
```

(continues on next page)

(continued from previous page)

```
IS 'MASKED WITH FUNCTION anon.fake_company()';

SECURITY LABEL FOR anon ON COLUMN customer.zipcode
IS 'MASKED WITH FUNCTION anon.random_zip()';

-- Performs static anonymization on the database.
SELECT anon.anonymize_database();

-- Anonymized data
SELECT * FROM customer;
 id  |     full_name      |   birth    |     employer      | zipcode | fk_shop
-----+--------------------+------------+-------------------+---------+---------
 911 | michel Duffus      | 1970-03-24 | Body Expressions  | 63824   | 12
 112 | andromach Tulip    | 1921-03-24 | Dot Darcy         | 38199   | 423
```

You can also use the `anon.anonymize_table()` and `anon.anonymize_column()` functions to anonymize a subset of the database (a specified table or column).

Dynamic masking

Dynamic masking refers to applying different masking rules for different users. For example, you can declare a role as "MASKED" to hide sensitive information from that role, while other user roles can still access the original data.

Dynamic masking hides sensitive information for specified users. Let's walk through an example.

1. Create and initialize the extension.

```
CREATE EXTENSION IF NOT EXISTS pgcrypto;
CREATE EXTENSION IF NOT EXISTS anon CASCADE;
SELECT anon.start_dynamic_masking();
-- Because this example uses faking functions, you need to run anon.init().
SELECT anon.init();
```

2. Create a table for testing.

```
CREATE TABLE people ( id TEXT, firstname TEXT, lastname TEXT, phone TEXT);
INSERT INTO people VALUES ('T1','Sarah', 'Conor','0609110911');
```

3. Define masking rules for the columns of the `people` table.

```
SECURITY LABEL FOR anon ON COLUMN people.lastname IS 'MASKED WITH FUNCTION anon.
fake_last_name()';
SECURITY LABEL FOR anon ON COLUMN people.phone IS 'MASKED WITH FUNCTION anon.
partial(phone,2,$$******$$,2)';
```

4. Create a new user and designate it as "MASKED".

```
CREATE ROLE skynet LOGIN;
alter ROLE skynet with password '123456';
SECURITY LABEL FOR anon ON ROLE skynet IS 'MASKED';
```

5. Switch to the new user to view the anonymized data.

```
\c - skynet
SELECT * FROM people;

-- The test result is as follows:
dynamic_masking=> SELECT * FROM people;
id | firstname | lastname |   phone
----+-----------+----------+------------
T1 | Sarah     | Watsica  | 06******11
(1 row)
```

As you can see, in addition to the general operations of creating and initializing the extension, the basic steps for dynamic masking are: define masking rules for the database object you want to anonymize (for example, a specific column), designate a user as "MASKED", and start dynamic masking. When this user logs in, the data they see will be anonymized.

7.7 Configure Row-Level and Column-Level Permissions

This document explains how to configure row-level security (RLS) and column-level security to achieve more fine-grained table access control, thereby restricting user access to specific data.

In a database security architecture, access control is implemented in layers. At a high level, SynxDB Elastic provides various security mechanisms to prevent unauthorized users from accessing the database cluster, such as host-based authentication, various authentication methods like LDAP or PAM, and limiting listening addresses.

Once an authorized user gains database access, security controls are applied at the database object level. By using the `GRANT` and `REVOKE` commands, combined with role-based access control, you can manage user permissions for specific database objects like tables and views.

This document focuses on more granular in-table security controls: how to configure permissions when a user has access to a table but business requirements dictate they cannot see specific rows or columns. In SynxDB Elastic, this is primarily achieved through two methods:

- Row-level security
- Column-level security

7.7.1 Row-level security

Row-Level Security (RLS) is a critical security feature in SynxDB Elastic. It allows the table owner to define security policies on a table to precisely control which rows different users can view and manipulate. You can think of an RLS policy as an automatically applied filter: when a user performs any operation on the table, the system first applies this policy, filtering the data rows visible to the user based on its defined rules.

RLS policies are powerful and flexible, allowing for access control based on specific commands (like `SELECT` or DML commands `INSERT`/`UPDATE`/`DELETE`), specific users or groups, or rows that meet certain conditions.

Characteristics of row-level security policies

- **Disabled by default**: By default, no RLS policies are defined on a table. As long as a user has SQL permissions, they can access all rows in the table.
- **Default-deny once enabled**: When the table owner enables RLS for a table using the `ALTER TABLE ... ENABLE ROW LEVEL SECURITY` command, the access policy defaults to "deny

all" . At this point, no user other than the table owner can access any rows in the table until an appropriate "allow" policy is created for them.

> **Note**
>
> Operations that apply to the entire table (such as `TRUNCATE` and `REFERENCES`) are not restricted by row-level security.

- **Policy flexibility**: Security policies can be set for specific commands, specific roles, or a combination of both. A policy can apply to all commands (`ALL`) or only to `SELECT`, `INSERT`, `UPDATE`, or `DELETE`. Policies can be granted to multiple roles and follow standard role membership and inheritance rules.
- **Privileges for superusers and owners**: Superusers and roles with the `BYPASSRLS` attribute are not subject to RLS policies. By default, the table owner is also not constrained, but policies can be forcibly applied to the owner using the `ALTER TABLE ... FORCE ROW LEVEL SECURITY` command.
- **Owner permissions**: Only the table owner can enable or disable RLS and add policies to the table.

Enable and create a security policy

Configuring RLS for a table involves two steps, both of which must be performed by the table owner: first, enable the RLS feature for the table, and second, create specific security policies.

1. Enable row-level security. Use the following command to enable RLS for the target table:

```
ALTER TABLE <table_name> ENABLE ROW LEVEL SECURITY;
```

2. Create a security policy. After enabling RLS, use the `CREATE POLICY` command to create specific policies. The core of a policy lies in the boolean expression defined in the `USING` or `WITH CHECK` clause. This expression is evaluated for each row before any conditions or functions from the user's query are executed. Only rows for which the expression returns `true` are visible or available for modification.
    - The `USING` expression controls which rows are visible to the user (for `SELECT`) or available for modification (for `UPDATE`, `DELETE`).

- The `WITH CHECK` expression controls which new rows can be added by `INSERT` or `UPDATE`.

Policy expressions run as part of the query and with the permissions of the user running the query. If you need to access data that the user does not have permission to, you can use `SECURITY DEFINER` functions.

The syntax for `CREATE POLICY` is as follows:

```
CREATE POLICY <name> ON <table_name>
    [ AS { PERMISSIVE | RESTRICTIVE } ]
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { <role_name> | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( <using_expression> ) ]
    [ WITH CHECK ( <check_expression> ) ]
```

Parameter descriptions:

- `name`: The name of the policy.
- `table_name`: The name of the table to which the policy applies.
- `PERMISSIVE`: Specifies the policy as permissive. All permissive policies applicable to a given query are combined using the boolean `OR` operator. Administrators can add to the set of accessible records by creating permissive policies. This is the default.
- `RESTRICTIVE`: Specifies the policy as restrictive. All restrictive policies applicable to a given query are combined using the boolean `AND` operator. Administrators can reduce the set of accessible records by creating restrictive policies, as each record must pass all restrictive policies.
- `FOR` clause: Specifies the command(s) to which the policy applies, such as `ALL`, `SELECT`, `INSERT`, `UPDATE`, `DELETE`.
- `TO` clause: Specifies the role(s) to which the policy applies. The default is `PUBLIC`, which applies the policy to all roles.
- `using_expression`: Any SQL conditional expression that returns a boolean value. The expression cannot contain any aggregate or window functions. For `SELECT`, `UPDATE`, or `DELETE` operations, only rows for which this expression returns `true` will be visible or modifiable.

- `check_expression`: Any SQL conditional expression that returns a boolean value, which also cannot contain aggregate or window functions. For `INSERT` or `UPDATE` operations, only new or updated rows for which this expression evaluates to `true` will be allowed. If the expression evaluates to `false` or `null` for any of the records, an error will be thrown.

Example: create a row-level security policy

The following example demonstrates how to set up a row-level security policy for a table, allowing users to access only data related to their department.

1. Connect to the database as an administrator: `psql -h <host_ip> -p <port> -U <user_name> -d <db_name>`.

2. Create a sample table and insert data:

```
CREATE TABLE projects (
    project_id SERIAL PRIMARY KEY,
    project_name TEXT,
    project_manager TEXT,
    department TEXT
);

INSERT INTO projects (project_name, project_manager, department) VALUES
('Project Alpha', 'john', 'Engineering'),
('Project Beta', 'kate', 'HR'),
('Project Gamma', 'bob', 'Sales');
```

3. Enable row-level security for the `projects` table:

```
ALTER TABLE projects ENABLE ROW LEVEL SECURITY;
```

4. Create a row-level security policy. This policy checks a session variable `myapp.current_department` to determine the user's department.

```
CREATE POLICY department_policy
ON projects
FOR SELECT
USING (department = current_setting('myapp.current_department'));
```

5. Create test users:

```
CREATE USER john WITH PASSWORD '<password>';
CREATE USER kate WITH PASSWORD '<password>';
CREATE USER bob WITH PASSWORD '<password>';
```

6. Grant the `john` user permission to query the `projects` table:

```
GRANT SELECT ON projects TO john;
```

7. Switch to the `john` user and set the department for the current session:

```
SET ROLE john;
SET myapp.current_department = 'Engineering';
```

8. Query the `projects` table as the `john` user. Because the department is set to 'Engineering', the query will only return projects from that department:

```
SELECT * FROM projects;

-- Expected result:
-- project_id | project_name  | project_manager | department
--------------+---------------+-----------------+-------------
--          1 | Project Alpha | john            | Engineering
```

7.7.2 Column-level security

Column-level security allows you to precisely control which columns in a table a user can access. By using the `GRANT` command, you can grant a user permission to view only specific columns, thereby protecting all other columns from access.

Column-level permissions and default privileges

In SynxDB Elastic, when certain types of objects are created, some permissions are granted to the `PUBLIC` role by default. However, for security, no privileges are granted to `PUBLIC` by default on tables, columns, sequences, foreign-data wrappers, foreign servers, large objects, schemas, or tablespaces.

For other object types, the default privileges granted to `PUBLIC` are as follows:

- **Databases**: `CONNECT` and `TEMPORARY` (create temporary tables) privileges.

- **Functions and procedures**: `EXECUTE` privilege.
- **Languages and data types**: `USAGE` privilege.

The object owner can `REVOKE` these default privileges. For maximum security, it is recommended to issue the `REVOKE` in the same transaction that creates the object; this leaves no window in which another user can use the object. You can also override these default privilege settings using the `ALTER DEFAULT PRIVILEGES` command.

The following table shows the abbreviations used for these privilege types in Access Control Lists (ACLs). You will see these letters in the output of psql commands or when viewing ACL columns in the system catalogs.

Privilege	Abbreviation	Applicable Object Types
`SELECT`	r ("read")	Large objects, sequences, tables (and table-like objects), columns
`INSERT`	a ("append")	Tables, columns
`UPDATE`	w ("write")	Large objects, sequences, tables, columns
`DELETE`	d	Tables
`TRUNCATE`	D	Tables
`REFERENCES`	x	Tables, columns
`TRIGGER`	t	Tables
`CREATE`	C	Databases, schemas, tablespaces
`CONNECT`	c	Databases
`TEMPORARY`	T	Databases
`EXECUTE`	X	Functions, procedures
`USAGE`	U	Domains, foreign-data wrappers, foreign servers, languages, schemas, sequences, types

Example: set column-level permissions

This example demonstrates how to grant a user access to a specific column, so they can only view the authorized column.

1. Create a test table and insert data:

```
CREATE TABLE t_user (n_id INT, c_name TEXT) TABLESPACE oss_test;
SET synxdb.warehouse TO wh0;
INSERT INTO t_user VALUES (1, 'john');
```

2. Create the user `john` and grant connect and usage permissions:

```
-- Creates a login role.
CREATE ROLE john LOGIN PASSWORD '123456';
```

(continues on next page)

(continued from previous page)

```
-- Creates a user mapping for john (if external tablespace access is needed).
CREATE USER MAPPING FOR john SERVER test_server OPTIONS (accesskey '..', secretkey
'..');

-- Grants privileges on the tablespace to john.
GRANT ALL PRIVILEGES ON TABLESPACE oss_test TO john;

-- Grants connect and usage permissions to john.
GRANT CONNECT ON DATABASE postgres TO john;
GRANT USAGE ON SCHEMA public TO john;
```

3. Grant the `SELECT` privilege on the `n_id` column of the `t_user` table to `john`:

```
GRANT SELECT (n_id) ON t_user TO john;
```

4. Query data as the `john` user:

```
-- Switches to the john user.
\c postgres john
You are now connected to database "postgres" as user "john".

-- Sets the warehouse for the session.
SET synxdb.warehouse TO wh0;

-- Querying the n_id column succeeds.
SELECT n_id FROM t_user;
-- n_id
--------
--    1
-- (1 row)

-- Querying the c_name column or all columns (*) will fail due to lack of
permission.
SELECT c_name FROM t_user;
-- ERROR:  permission denied for table t_user

SELECT * FROM t_user;
-- ERROR:  permission denied for table t_user
```

Chapter 8

Developer Guides

8.1 Use JDBC

SynxDB Elastic is a cloud-native database based on the Greenplum/PostgreSQL 14.4 kernel, featuring a separation of storage and compute. Because it is compatible with Greenplum and PostgreSQL, JDBC access methods are the same as those for Greenplum/PostgreSQL.

This guide explains how to connect to SynxDB Elastic using JDBC and perform database operations.

8.1.1 Prerequisites

Before connecting to SynxDB Elastic via JDBC, ensure you have:

- A Java runtime environment (JDK 1.8 or later) installed.
- The PostgreSQL JDBC driver (`postgresql-<version>.jar`) downloaded.
- Connection details for SynxDB Elastic, including host, port, database name, username, and password.

8.1.2 Step 1. Download the JDBC driver

The JDBC driver can be downloaded from the official PostgreSQL website. Use the latest stable version compatible with SynxDB Elastic.

Example download command (for version 42.5.0):

```
wget https://jdbc.postgresql.org/download/postgresql-42.5.0.jar
```

8.1.3 Step 2. Connect to SynxDB Elastic

To connect to SynxDB Elastic, use the following connection string format in you Java program.

```
jdbc:postgresql://<host>:<port>/<database>?parameters
```

Common parameters:

- `user=<username>`: Specifies the database username.
- `password=<password>`: Specifies the database password.
- `ssl=<true|false>`: Enables or disables SSL connection.
- `ApplicationName=<app_name>`: Optional, used to identify the client application.

Example:

```
jdbc:postgresql://db.example.com:5432/mydb?user=myuser&password=mypass&ssl=true
```

The following example demonstrates how to connect to SynxDB Elastic using JDBC:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class SynxDBElasticExample {
   public static void main(String[] args) {
      String url = "jdbc:postgresql://your-synxdb-host:5432/your_database";
      String user = "your_username";
      String password = "your_password";
```

(continues on next page)

(continued from previous page)

```
        try (Connection conn = DriverManager.getConnection(url, user, password);
             Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery("SELECT version();")) {

            while (rs.next()) {
               System.out.println("Database Version: " + rs.getString(1));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

8.1.4 Step 3. Set up the environment

Download PostgreSQL JDBC driver

Before connecting to the database, download the PostgreSQL JDBC driver:

```
wget https://jdbc.postgresql.org/download/postgresql-42.5.0.jar
```

Compile Java code

After writing the Java program, compile it using the downloaded JDBC driver:

```
javac -cp postgresql-42.5.0.jar YourJavaProgram.java
```

Run the Java program

Execute the Java program with the classpath set to include the JDBC driver:

```
java -cp .:postgresql-42.5.0.jar YourJavaProgram
```

8.1.5 Execute SQL statements

To execute SQL statements through JDBC, you can refer to the following sections to add code to your java program.

Query data

```
String query = "SELECT id, name FROM users";
try (Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(query)) {
   while (rs.next()) {
      int id = rs.getInt("id");
      String name = rs.getString("name");
      System.out.println("ID: " + id + ", Name: " + name);
   }
}
```

Insert data

```
String insertSQL = "INSERT INTO users (id, name) VALUES (1, 'Alice')";
try (Statement stmt = conn.createStatement()) {
   int rowsAffected = stmt.executeUpdate(insertSQL);
   System.out.println("Rows inserted: " + rowsAffected);
}
```

Update data

```
String updateSQL = "UPDATE users SET name = 'Bob' WHERE id = 1";
try (Statement stmt = conn.createStatement()) {
   int rowsAffected = stmt.executeUpdate(updateSQL);
   System.out.println("Rows updated: " + rowsAffected);
}
```

Delete data

```
String deleteSQL = "DELETE FROM users WHERE id = 1";
try (Statement stmt = conn.createStatement()) {
   int rowsAffected = stmt.executeUpdate(deleteSQL);
   System.out.println("Rows deleted: " + rowsAffected);
}
```

8.1.6 Transaction management

JDBC allows explicit transaction control:

```
conn.setAutoCommit(false);
try (Statement stmt = conn.createStatement()) {
   stmt.executeUpdate("INSERT INTO users (id, name) VALUES (2, 'Charlie')");
   stmt.executeUpdate("UPDATE users SET name = 'Charlie Updated' WHERE id = 2");
   conn.commit();
} catch (SQLException e) {
   conn.rollback();
   e.printStackTrace();
}
```

8.1.7 Use connection pools

In production environments, using a connection pool (for example, HikariCP) improves performance.

Add HikariCP dependency

For Maven:

```
<dependency>
   <groupId>com.zaxxer</groupId>
   <artifactId>HikariCP</artifactId>
   <version>5.0.1</version>
</dependency>
```

Configure HikariCP

```
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:postgresql://your-synxdb-host:5432/your_database");
config.setUsername("your_username");
config.setPassword("your_password");
config.setMaximumPoolSize(10);
HikariDataSource dataSource = new HikariDataSource(config);
```

8.1.8 Troubleshoot connection issues

If the connection fails, check:

- Network accessibility to SynxDB Elastic.
- Firewall or security group settings allowing PostgreSQL port (default 5432).
- Correct JDBC URL, username, and password.
- Database logs for detailed error messages.

8.2 Use ODBC

SynxDB Elastic is a cloud-native database based on the Greenplum/PostgreSQL 14.4 kernel, featuring a separation of storage and compute. Because it is compatible with Greenplum and PostgreSQL, ODBC access methods are the same as those for Greenplum/PostgreSQL.

This guide explains how to connect to SynxDB Elastic using ODBC on Linux or macOS and perform database operations.

8.2.1 Prerequisites

Before connecting to SynxDB Elastic via ODBC, ensure you have:

- The PostgreSQL ODBC driver installed.
- An ODBC-compliant application or tool (for example, `psqlODBC`, `isql`, BI tools like Tableau).
- Connection details for SynxDB Elastic, including host, port, database name, username, and password.

8.2.2 Install the ODBC driver

The PostgreSQL ODBC driver can be downloaded from the official PostgreSQL ODBC website. Ensure you install the correct version for your operating system.

1. Install the required packages (for Ubuntu/Debian):

```
sudo apt update && sudo apt install -y unixodbc odbc-postgresql
```

2. Verify installation:

```
odbcinst -q -d
```

   The output should list `PostgreSQL` as a supported driver.

8.2.3 Configure ODBC connection

Edit the ODBC driver configuration file (`/etc/odbcinst.ini`):

```
[PostgreSQL]
Description = ODBC for PostgreSQL
Driver = /usr/lib/x86_64-linux-gnu/odbc/psqlodbcw.so
Setup = /usr/lib/x86_64-linux-gnu/odbc/libodbcpsqlS.so
```

Edit the ODBC data source file (`/etc/odbc.ini`):

```
[SynxDBElastic]
Driver = PostgreSQL
ServerName = your-synxdb-host
Port = 5432
Database = your_database
UserName = your_username
Password = your_password
SSLmode = prefer
```

8.2.4 Connect to SynxDB Elastic via ODBC

Use `isql`

Once the DSN is configured, you can test the connection using `isql`:

```
isql -v SynxDBElastic your_username your_password
```

If the connection is successful, you will enter an interactive SQL shell.

Use Python (`pyodbc`)

If you prefer to connect using Python, install the `pyodbc` package and use the following script:

```
import pyodbc

conn = pyodbc.connect(
   "DSN=SynxDBElastic;UID=your_username;PWD=your_password"
)
cursor = conn.cursor()
cursor.execute("SELECT version();")
row = cursor.fetchone()
```

(continues on next page)

(continued from previous page)

```
print("Database Version:", row[0])
conn.close()
```

8.2.5 Execute SQL statements

Query data

```
cursor.execute("SELECT id, name FROM users;")
for row in cursor.fetchall():
   print(f"ID: {row[0]}, Name: {row[1]}")
```

Insert data

```
cursor.execute("INSERT INTO users (id, name) VALUES (1, 'Alice');")
conn.commit()
```

Update data

```
cursor.execute("UPDATE users SET name = 'Bob' WHERE id = 1;")
conn.commit()
```

Delete data

```
cursor.execute("DELETE FROM users WHERE id = 1;")
conn.commit()
```

8.2.6 Connection troubleshooting

If the connection fails, check:

- Network accessibility to SynxDB Elastic.
- Firewall or security group settings allowing PostgreSQL port (default 5432).
- Correct ODBC DSN settings.
- Database logs for error messages.

8.3 Supported BI tools

SynxDB Elastic is a cloud-native, storage-compute-separated database built on the Apache Cloudberry (Greenplum) kernel.

Any BI tool that supports Greenplum is also compatible with SynxDB Elastic. This includes, but is not limited to:

- Tableau
- Power BI
- Looker
- Apache Superset
- Metabase
- Qlik Sense

To set up and configure your preferred BI tool, follow the connection methods of *JDBC driver* and *ODBC driver*.

For further assistance, consult the documentation of your BI tool.

Chapter 9

Tutorials

9.1 Quick-Start Guide

This guide is designed to help you quickly get started with SynxDB Elastic. By following this guide, you will learn how to access the SynxDB Elastic console, create necessary resources (such as accounts, users, and warehouses), run SQL queries via the client or console, load external data, and scale clusters.

SynxDB Elastic is a cloud-native, distributed analytical database that offers enterprises efficient, flexible, and scalable solutions for data management and analysis. It supports high-performance SQL queries, making it ideal for large-scale data analysis tasks. This guide aims to provide you with clear, step-by-step instructions, ensuring that even first-time users can easily complete the basic operations and lay a foundation for more advanced learning.

9.1.1 Access the console and log in

Before creating resources and running queries in SynxDB Elastic, you need to access and log into the console. Follow these steps to get started:

1. Open your browser and access the console login page.
2. Fill in the organization name, user name, and password.
3. Click **Login**.

After login, you are now ready to start creating resources.

9.1.2 Create resources

Before you start querying sample data, you need to create a few basic resources, including:

- Account: The basic organizational unit in SynxDB Elastic, and all subsequent operations will depend on it.
- User: Required to connect to and manage databases.
- Warehouse: The core resource for running SQL queries. Each warehouse can contain multiple segment nodes.

Create an account

1. Click the avatar in the top-right corner, and then click **Accounts** in the menu to enter the account page.
2. Click **Create Account** in the top-right corner to open the **Create Account** dialog.
3. Enter an account name, choose the cloud provider and region. Then click **OK**.
4. Return to the **Accounts** page. Click **Switch account** in the top-right corner and select the newly created account from the dropdown list.

Once the account is created, you can go on creating other resources.

Create a user

1. Click the SynxDB logo in the top-left corner of the console to access the **Dashboard** page.
2. Click **Users** in the left navigation bar to access the user page.
3. Click **+ Create User** in the top-right corner to open the user creation dialog.
4. Enter a user name and password, select a database role (the **login** role is mandatory), and click **OK**.

Once completed, the newly created user and its role information will appear in the user list.

Create a warehouse

1. In the left navigation bar, click **Warehouses** to access the warehouse page.
2. Click **+ Create Warehouse** in the top-right corner to open the warehouse creation dialog.
3. Enter a warehouse name (for example `warehouse1`), set the number of segments (2 is recommended), select the previously created user, and click OK.
4. Return to the **Warehouses** page and wait for a moment. Refresh the page until the warehouse status shows **Running**.

Once the warehouse is created, you are ready to connect to the database.

9.1.3 Connect to the database

After creating the necessary resources, you can connect to the SynxDB Elastic database using the psql client. Open a terminal and run the following command (replace the placeholders with your values), then enter the password you have set when creating the user:

```
psql -h <host_address> -U <user> -d postgres
```

Tip

- To get `<host_address>`, click the avatar in the top-right corner of the console and select **Accounts**. On the **Accounts** page, find the host address associated with your account.
- To get `<user>`, click **Users** in the left navigation bar of the console. Find the target user name on the user list page.
- `postgres`: A built-in database created with the data warehouse.

Once connected successfully, you can start executing SQL queries.

9.1.4 Query sample data

After connecting to the database, you can explore SynxDB Elastic by querying, creating tables, and inserting data. This section introduces two methods: querying data using a client or using the SQL editor in the console.

Method 1: Use a client

1. Open the terminal. Connect to the SynxDB Elastic database via the `psql` client.
2. Check the current tables in the database:

```
\dt
```

3. Before executing SQL operations, bind the task to a warehouse `warehouse1`:

```
SET warehouse TO warehouse1;
```

4. Create a sample table employees to store sample data:

```
CREATE TABLE employees (
   id SERIAL,
   name VARCHAR(50),
   position VARCHAR(50),
   salary NUMERIC(10, 2)
);
```

5. Insert sample data into the employees table:

```
INSERT INTO employees (name, position, salary) VALUES ('Alice', 'Engineer',
120000.00);
```

6. Query the table data:

```
SELECT * FROM employees;
```

Method 2: Use the worksheet in the console

If you prefer operations on a graphical interface, you can use the built-in SQL editor in the SynxDB Elastic console to execute SQL operations. The built-in SQL editor allows you to operate the database quickly and view results intuitively.

1. In the left navigation bar of the console, click **Worksheets**.
2. In the worksheet list, click worksheet to open the built-in SQL editor.
3. Click the **Select a warehouse** dropdown menu and select `warehouse1` you have created.

4. In the editor, enter and execute the following SQL statement to create the employees table:

```
CREATE TABLE employees (
   id SERIAL,
   name VARCHAR(50),
   position VARCHAR(50),
   salary NUMERIC(10, 2)
);
```

5. Insert sample data into the `employees` table:

```
INSERT INTO employees (name, position, salary) VALUES ('Alice', 'Engineer',
120000.00);
```

6. Query the table data:

```
SELECT * FROM employees;
```

9.1.5 Load and query external data

You can load sample data from cloud object storage into SynxDB Elastic by following these steps.

1. Create a file named `employees.csv` on your local machine, and add the following sample data to the file:

```
employee_id,first_name,last_name,department_id,hire_date
101,John,Doe,10,2020-01-15
102,Jane,Smith,20,2019-03-22
103,Emily,Davis,10,2021-07-01
104,Michael,Brown,30,2018-11-05
105,Sarah,Johnson,20,2022-02-14
```

2. Upload the `employees.csv` file to the Amazon S3 service in the `us-west-1` region. Set the bucket policies correctly, granting appropriate permissions to ensure accessibility. Refer to the Amazon S3 user documentation for detailed instructions.

3. Open the SynxDB Elastic console and navigate to the SQL editor, or connect to SynxDB Elastic using the `psql` client.

4. Execute the following statement to create a foreign data wrapper:

```
CREATE FOREIGN DATA WRAPPER datalake_fdw
HANDLER datalake_fdw_handler
VALIDATOR datalake_fdw_validator
OPTIONS ( mpp_execute 'all segments' );
```

5. Create a foreign server. When executing the following statement, replace `<hostname>` with the object storage host information (for example, `https://<bucket>.<region>.s3.amazonaws.com`) and `<protocol>` with the protocol for the cloud platform s3.

```
CREATE SERVER foreign_server
FOREIGN DATA WRAPPER datalake_fdw
OPTIONS (host '<hostname>', protocol '<protocol>', isvirtual 'false',ishttps 'true
');
```

6. Create a user mapping. When executing the following statement, replace `<access_key>` and `<secret_key>` with the actual credentials. Refer to Amazon S3 Documentation - Identity and Access Management for Amazon S3 for obtaining these credentials.

```
CREATE USER MAPPING FOR gpadmin SERVER foreign_server
OPTIONS (user 'gpadmin', accesskey '<access_key>', secretkey '<secret_key>');
```

7. Create the external table. When executing the following statement, replace `<file_path>` with the actual path to the `employees.csv` file in object storage (for example, `/example-cloud-doc/`).

```
CREATE FOREIGN TABLE example(
   employee_id INT,
   first_name VARCHAR(50),
   last_name VARCHAR(50),
   department_id INT,
   hire_date DATE)
SERVER foreign_server
OPTIONS (filePath '<file_path>', compression 'none', enableCache 'false', format
'csv');
```

8. Now you can query the data in the external table:

```
SELECT * FROM example;
```

9.1.6 Explore the unique advantages of the data warehouse

Multiple warehouses accessing shared storage

SynxDB Elastic supports shared data storage across multiple warehouses. For instance, tables created in warehouse1 can be accessed by warehouse2. Follow these steps to explore this feature:

1. Create a new warehouse named `warehouse2` by following the steps for creating a warehouse.
2. Open the built-in SQL editor by following the steps for querying data using the console.
3. Click the `Select a warehouse` dropdown menu and select `warehouse2`.
4. Query the `employees` table created on `warehouse1` from `warehouse2`:

   ```
   SELECT * FROM employees;
   ```

   You can query the table data created on `warehouse1` directly from the `warehouse2` cluster.

Warehouse elastic scaling-in and scaling-out

You can elastically scale warehouses to meet dynamic business workloads. Here is an example of scaling `warehouse1`:

1. Open the SynxDB Elastic console and click **Warehouses** in the left navigation bar to access the warehouse page.
2. Locate `warehouse1` and click **Edit Warehouse** in its row.
3. Adjust segment count in the **Edit Warehouse** dialog. Increase the value for scaling out or decrease it for scaling in based on your requirements.
4. Click **OK** and wait for the changes to take effect.

9.1.7 Summary

Congratulations! In this quick-start guide, you have completed the following tasks:

- Registered and accessed the SynxDB Elastic console.
- Created essential resources (accounts, users, and warehouses).

- Run SQL queries using a client or the console.
- Loaded and queried external data.
- Explored the unique advantages of SynxDB Elastic, such as shared storage and elastic scaling.

These steps have introduced you to the core features of SynxDB Elastic and laid a foundation for further exploration and usage.

9.2 Perform TPC-DS Benchmark Testing

This guide describes how to run the TPC-DS benchmark on SynxDB Elastic v4.x.

9.2.1 Prerequisites

Ensure the following conditions are met before running the test from a remote client host:

- `psql` client is installed and passwordless access to the remote cluster is configured (via `.pgpass`).
- The `gpadmin` database is created with the default warehouse parameter:

```
ALTER ROLE gpadmin SET warehouse = 'testforcloud';
```

  Replace `gpadmin` and `testforcloud` with the real user and warehouse you are using.

9.2.2 Step 1. Install TPC-DS tools dependencies

Install the required dependencies on `mdw`:

```
ssh root@mdw
yum -y install gcc make byacc
```

The original source code is available at: http://tpc.org/tpc_documents_current_versions/current_specifications5.asp.

9.2.3 Step 2. Download and install TPC-DS tools

Clone the TPC-DS tools repository from GitHub:

```
ssh gpadmin@mdw
git clone https://github.com/SynxDBFE/TPC-DS-CBDB.git
```

9.2.4 Step 3. Get psql connection options

To run the benchmark, get the necessary connection options for PostgreSQL (`psql`).

1. Log in to the Kubernetes (K8s) cluster and identify the tenant namespace:

```
kubectl get ns
```

Example output:

```
NAME                                          STATUS   AGE
default                                       Active   97d
synxdb-system-4x                              Active   96d
kube-node-lease                               Active   97d
kube-public                                   Active   97d
kube-system                                   Active   97d
storage-system                                Active   97d
tenant2-313a164c-cfe9-4514-941d-4c1d548ddf97   Active   93d
```

2. Obtain the cluster mapping ports for the tenant:

```
kubectl get svc -o wide -n tenant2-313a164c-cfe9-4514-941d-4c1d548ddf97
```

Example output:

```
NAME                    TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE   SELECTOR
cloudberry-proxy        ClusterIP   None           <none>        5432/TCP
93d   app=cloudberry-proxy
cloudberry-proxy-node   NodePort    10.96.199.21   <none>        5432:32355/TCP
93d   app=cloudberry-proxy
```

In this example, the obtained port is `32355`, and the IP can be any node in the K8s cluster (for example, `10.14.3.242`).

9.2.5 Step 4. Modify configuration files

Edit the `tpcds_variables.sh` file to configure the test parameters. Fill the `PSQL_OPTIONS` with the real psql connection information you have obtained. Edit `GEN_DATA_SCALE` according to your cluster size and need.

```
ssh gpadmin@mdw
cd ~/TPC-DS-CBDB
vim tpcds_variables.sh
```

Example configuration:

```
export RUN_MODEL="cloud"
export RANDOM_DISTRIBUTION="true"
export TABLE_STORAGE_OPTIONS="compresstype=zstd, compresslevel=5"
export CLIENT_GEN_PATH="/tmp/dsbenchmark"
export PSQL_OPTIONS="-h 10.14.3.242 -p 32355"
export GEN_DATA_SCALE="10"  # Generate 10GB of data.
export MULTI_USER_COUNT="5"   # Number of concurrent streams for the throughput run.
export RUN_SINGLE_USER_REPORTS="true"
export RUN_MULTI_USER="true"
export RUN_MULTI_USER_QGEN="true"
export RUN_MULTI_USER_REPORTS="true"
export RUN_SCORE="true"
```

For additional configuration parameters, refer to the repository README file: https://github.com/SynxDBFE/TPC-DS-CBDB/blob/main/README.md.

9.2.6 Step 5. Run the TPC-DS Benchmark

To execute the benchmark, log in as `gpadmin` and run the test script:

```
ssh gpadmin@mdw
cd ~/TPC-DS-CBDB
./run.sh
```

Chapter 10

Reference Guide

10.1 SQL Syntax

SynxDB Elastic is built on Apache Cloudberry as its database engine. Therefore, its SQL syntax, system functions, and data types are highly compatible with Apache Cloudberry. Users can refer to the Apache Cloudberry SQL statement reference for detailed syntax and usage.

However, SynxDB Elastic introduces several architectural and functional changes compared to Apache Cloudberry. As a result, while the SQL syntax remains highly compatible, actual behaviors might differ. Users should always refer to SynxDB Elastic documentation and validate their queries based on the product's actual execution behaviors. Some key differences include:

- Index management: SynxDB Elastic does not support the `CREATE INDEX` statement, and index-related DDL statements are unavailable.
- Resource management: SynxDB Elastic uses a container-based resource management system, which differs from Apache Cloudberry's resource queue mechanism.
- Separation of storage and compute: Certain DDL, DML, and query optimization strategies have been adjusted to align with the storage-compute separation architecture, leading to behavioral differences from Apache Cloudberry.

10.1.1 UNDROP TABLE

`UNDROP TABLE` restores a dropped table from the recycle bin.

Synopsis

```
UNDROP TABLE <new_table_name> AS <recycle_bin_table_name>;
```

Description

When a table is dropped, you can restore it within the data retention period using the `UNDROP TABLE` command. Before restoring, you must first query the system table `pg_recycle_bin` to find the unique name of the table in the recycle bin.

Parameters

- `<new_table_name>`: The new name for the restored table.
- `<recycle_bin_table_name>`: The name of the table to be restored, as it exists in `pg_recycle_bin`.

Examples

```
-- Assume table t1 was accidentally dropped.
DROP TABLE t1;

-- 1. Query the recycle bin for the unique name of the dropped table.
SELECT relname FROM pg_recycle_bin WHERE orig_relname = 't1';

-- Assume the query result is 't1_1678886400'.

-- 2. Use the retrieved name to restore the table and name it t1.
UNDROP TABLE t1 AS "t1_1678886400";
```

See also

Time travel and flashback recovery